Refinements to the Analysis and Synthesis of Musical Tones using the Karhunen-Loève Transform

by

William Brent Weeks
B.Eng., University of Victoria, 1989

ACCEPTED
TY OF GRADUATE STUDIES

DEAN

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming to the required standard

[redacted]

Dr. Vijay K. Bhargava, Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Dr. R. Lynn Kirlin, Co-Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Mr. Michael M. Longton, Outside Member
(Department of Music)

[redacted]

Dr. W. Andrew Schloss, Outside Member
(Department of Music)

[redacted]

Dr. Sherman R. Waddell, External Examiner
(Department of Physics and Oceanography,
Royal Roads Military College)

© WILLIAM BRENT WEEKS, 1992
University of Victoria

All rights reserved. Thesis may not be reproduced in whole or in part,
by photocopy or other means, without the permission of the author.

Supervisor: Dr. V.K. Bhargava
Co-Supervisor: Dr. R.L. Kirlin

ABSTRACT

A procedure for the analysis and synthesis of musical tones using the Karhunen-Loève Transform is reviewed. Analysis of a well-correlated set of musical instrument tones yields a small number (typically less than or equal to four) of wavetable oscillators which can be used for re-synthesis of the analyzed tones. Some refinements to the analysis procedure are introduced. Results of applying the refined analysis procedure to tracking variations in individual instrument timbres are presented. Some instruments exhibit large timbral variations over their performance range and are not tracked well by a single set of at most four wavetable oscillators. Quality re-synthesis has been achieved for most instruments analyzed; however, a number of consistent deficiencies are audible in the synthesized tones. In particular, many synthesized tones exhibit a buzz during their decay, and noise-like components of tones are not synthesized well. Improvements to address these deficiencies as well as certain limitations in the analysis procedure are suggested. Finally, a synthesizer structure incorporating some of these improvements is proposed.

Examiners:

[redacted]

Dr. Vijay K. Bhargava, Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Dr. R. Lynn Kirlin, Co-Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Mr. Michael M. Longton, Outside Member
(Department of Music)

[redacted]

Dr. W. Andrew Schloss, Outside Member
(Department of Music)

[redacted]

Dr. Sherman R. Waddell, External Examiner
(Department of Physics and Oceanography,
Royal Roads Military College)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would first like to acknowledge the financial support of both NSERC and the University of Victoria during the course of my research. Besides financial support, many people have helped along the way. First I wish to thank the members of my supervisory committee, Professors Bhargava, Kirlin, Longton and Schloss, for the guidance, encouragement and consultation they have provided throughout the course of my research. I also wish to thank Professor Peter F. Driessen for bringing to my attention the work of Stapleton and Bass on KL-based analysis/synthesis which has provided the foundation for this thesis. It would be remiss not to mention the help I often received from computer and technical support staff, in particular Mark McIntosh, Roger Kelly and Al Keddy. As well, numerous fellow graduate students have provided invaluable assistance with a variety of problems (mostly related to using computers) along the way. In particular, I wish to thank Brad Hedstrom. I also appreciate the help Doug Peters provided in proof-reading this thesis. There are also a number of people to thank from outside the university. First I wish to thank my employers at IVL Technologies, Phil Scott and Brian Gibson, for their willingness to let me start work on a part-time basis while completing this thesis. Second, I wish to thank both family and friends who have encouraged me to complete the task. Of these, I should first thank my mother, Edith Chown, who by returning to school herself, helped to spur me on. Second, I wish to thank my wife, Shelley, for her ongoing love and support. Finally, I wish to thank my Creator and Redeemer, the Lord Jesus Christ, for daily giving me both the life and strength to work and learn.

To Shelley Lynne,

my "Special Lady"

CHAPTER 1
INTRODUCTION

Archaeological evidence indicates that music making has been a part of human societies since the earliest known cultures and civilizations. Integral to the making of music has been the development of musical instruments. According to Yehudi Menuhin, "the creation of musical instruments is one of the great human miracles." [1, p. 8] Early instruments were fashioned from bone, horn, shell, wood, stone, natural fibers, animal gut and skins. Some instruments imitated the sounds of nature, others produced new and distinctive tones.

Today, we are still exposed to many instruments which resemble these simple beginnings, such as the pan flute or recorder. In addition, technological developments have resulted in more sophisticated families of instruments, such as the brass or woodwind sections of a modern orchestra. The most recent development to have a significant impact on the making of musical instruments has been the growth of electronics and computer technology. Out of this technology has emerged a new class of instruments: electronic synthesizers.

1.1. Electronic Synthesizers - A Survey

Previous to electronic synthesizers, all musical instruments were acoustic in nature. Acoustic musical instruments directly produce vibrations in the air which are audible to humans. For example a snare drum is struck by a stick, resulting in the drum skin and metal snare vibrating, which in turn results

in audible air vibrations. A stringed instrument such as the violin produces sound as a result of a string being excited by a bow. Woodwind instruments, such as the clarinet, acoustically amplify and resonate in response to the sound of a vibrating reed, which is excited by the air pressure of a player blowing. Brass instruments are similar to woodwinds, with finger holes replaced by valves and the reed replaced by the player's lips. A modern pipe organ, with the aid of powered pumps, blows air through its various pipes.

In all of the above cases, some physical mechanism vibrates at audio frequencies, and these vibrations cause audible vibrations in the air. Acoustic instruments can be distinguished by the physical means used to produce acoustic vibrations. In contrast, electronic synthesizers all use the same basic means of producing acoustic vibrations - the loudspeaker. There is also a third class of hybrid acoustic/electronic instruments. The electric guitar and electric bass fall into this category. Both produce acoustic vibrations, but at a very low energy level. These vibrations are sensed by transducers called pickups, then amplified and possibly transformed before being output through a loudspeaker.

Loudspeakers all operate on the same principle. They translate electrical vibrations (fluctuating voltage or current) into mechanical vibrations which result in acoustic vibrations. An electronic synthesizer produces sound by simply generating electrical vibrations which are fed to a loudspeaker. It is the many schemes of generating electrical vibrations which distinguish electronic synthesizers.

There is one final and important feature which distinguishes electronic synthesizers from acoustic instruments. This feature is the means of playing the instrument, or in computer terminology, the user interface. The playing

technique for any given acoustic instrument is directly tied to the means of producing sound. The means of playing cannot be abstracted from the sound generated for an acoustic instrument. While a restriction, this is also an important feature. The direct relationship between playing technique and resulting tone for many acoustic instruments affords a wide range of subtlety and nuance which can be used to great advantage by those who master the instruments.

In contrast, modern electronic synthesizers can support abstracting the means of playing from the generation of sound. Electronic synthesizers typically utilize a keyboard interface similar to a piano or organ, but this need not be the case. New interface devices are being developed, such as the Mathews/Boie Radio Drum [2]. The MIDI (Musical Instrument Digital Interface) standard is based on abstracting performance parameters from sound generation parameters [3]. The abstraction of performance and playing technique from sound generation allows the designer of an electronic synthesizer to largely ignore the question of how the instrument will be played, while concentrating on the method of generating musically useful sounds.

Regardless of the means of performance, electronic synthesizers are typically used for two purposes. Originally, electronic synthesizers were used to create new musical sounds and special effects. This application continues to be relevant. As well, efforts were made to emulate known sounds or musical tones. With recent advances in technology, this latter application has become very popular. It is now possible to purchase a synthesizer which is truly a "one-man band." Such a synthesizer can emulate not only traditional Western orchestral and popular instruments, but also instruments from

other cultures, resulting in a one-man, "multi-ethnic" band. Synthesizers provide a powerful tool for today's performer/composer.

1.1.1. Synthesizer Classification by Hardware

It is possible to group synthesizers based on two different classification criteria. One criterion is the electronic hardware used to produce sound. Early synthesizers were based on analog circuits. The fundamental building blocks of analog synthesizers are oscillators, noise generators, modulators, filters, amplifiers and gates. An analog synthesizer is really just an analog computer designed especially for producing sounds. Early analog synthesizers were programmed by physically interconnecting different modules with patch cords and setting various knobs, sliders and switches. The use of the term "patch" for a specific synthesizer configuration comes from these early synthesizers. A disadvantage of early synthesizers was that only one configuration of the synthesizer was available at any one time. Saving the settings for a configuration required writing down all the interconnections and knob/switch settings. A significant amount of time was required to tear down and set up a new patch. Analog synthesizers can produce many interesting sounds, but are not able to emulate many acoustic instruments. Until recently, all electronic organs were just analog synthesizers with a set of preset configurations and a small number of variable control knobs and sliders.

Another group of early synthesizers was developed on digital computers. Until recently, these computer-based synthesizers did not run in real-time. Computer-based synthesis (commonly referred to as software synthesis) involves writing a sound generation program, then running the program to

output a discrete-time sampled waveform. With early digital computers, the output of a sound generation program was written to tape for audio playback using a digital-to-analog converter (DAC). As well, acoustic instruments could be recorded on tape and then read into the computer using an analog-to-digital converter (ADC). Access to discrete-time sampled acoustic waveforms allows writing computer programs to analyze acoustic instrument tones or other natural sounds. Synthesis programs can use results from analysis programs. The great advantage of computer-based synthesis is the generality afforded in specifying a music/sound generation algorithm. Algorithms can be designed to an arbitrary degree of complexity. The disadvantage of computer-based synthesis is the typically large amount of time required to produce a comparatively short segment of sound. Computer-based synthesis continues to be an active field of work [4]. A thorough overview of computer-based synthesis using the *cmusic* program is provided in Moore's book [5].

With the growth of digital electronics technology, the "patch"-saving problem of analog synthesizers was overcome by no longer using physical wires to interconnect modules, but using digitally controlled switches. The settings of all the interconnection switches as well as other knobs, sliders and switches could be stored in digital memory. A single synthesizer with memory could store many individual configurations or patches. A further improvement was to control the frequency of oscillators digitally. This overcame tuning problems inherent in earlier analog synthesizers.

More recently, synthesizers have been built entirely around digital circuits, except for DAC's and amplifiers at the outputs. These synthesizers either emulate the workings of an analog synthesizer or implement the less

computation-intensive computer-based synthesis algorithms. Essentially, digital synthesizers are highly specialized digital computers designed to generate musical sounds. Typically they are oriented to one specific synthesis algorithm or use hybrids of two or more algorithms. The most recent development in hardware engines for digital synthesis are music workstations. An example is the IRCAM/Ariel music workstation, built around a NeXT computer. It provides full control over programming different algorithms, just as in the early use of computers, but allows even reasonably complicated algorithms to run in real-time [6]. With this level of technology, it is really the different synthesis algorithms which distinguish synthesizer implementations, as opposed to the underlying electronic hardware.

1.1.2. Synthesizer Classification by Algorithm

The majority of musical tone synthesis algorithms have only been implemented on discrete-time based digital synthesizers. Only a small number of algorithms have been implemented on analog synthesizers, and any of these can be implemented in a digital scheme as well. A number of papers have been written to introduce or survey the multitude of algorithms which have been employed in digital synthesis. Of note are the papers by De Poli [7] and Gordon [8]. A more recent paper by Smith [9] classifies digital synthesis algorithms into four reasonably distinct categories: processed recording, spectral modeling, physical modeling and abstract algorithms. While these categories are not mutually exclusive, they do serve as a meaningful way to divide up the many synthesis algorithms which have been developed. In the following, we will not attempt to exhaustively list all

known synthesis algorithms, but instead highlight some of the more significant algorithms in each category.

Processed Recording

In its most "brute-force" implementation, processed recording simply involves playing back pre-recorded samples. Many of today's most popular synthesizers use this sample-playback technique, striving to realistically reproduce acoustic instrument tones. Pre-recorded samples can be shortened or looped for arbitrary duration, scaled in amplitude to the appropriate dynamic level, and pitch-shifted to a desired pitch. It is typically necessary to record several samples of an instrument's tone at varying pitch and possibly varying dynamic level to adequately represent the full range of an instrument with a particular performance technique. (Many instruments have more than one playing or performance technique. For example, the violin has three performance techniques: bowed string, struck string or marcato, and plucked string or pizzicato.)

In Figure 1.1, nine different trumpet tone samples are shown. For each note, A3 (220 Hz), G4 (392 Hz) and F5 (698.5 Hz), the trumpet was played at three different dynamic levels, *p* (*piano*, or soft), *mf* (*mezzo-forte*, or medium loud) and *f* (*forte*, or loud). It is clear from the figure that for differing pitches (in this case, approximately one octave apart), the typical waveform for a trumpet noticeably changes. Waveform variation with volume at a constant pitch is also discernable, though to a much lesser extent. For a trumpet, a note is not only louder, but "brighter" when played at a louder dynamic level. This is due to more high-frequency components being excited when the instrument is blown harder [5]. Instead of looking at the time-domain waveform, one can also look at the frequency spectra of the tones. A thorough

analysis of frequency spectra (or timbral) variation with pitch for standard orchestral instruments has been performed by Sandell [10].

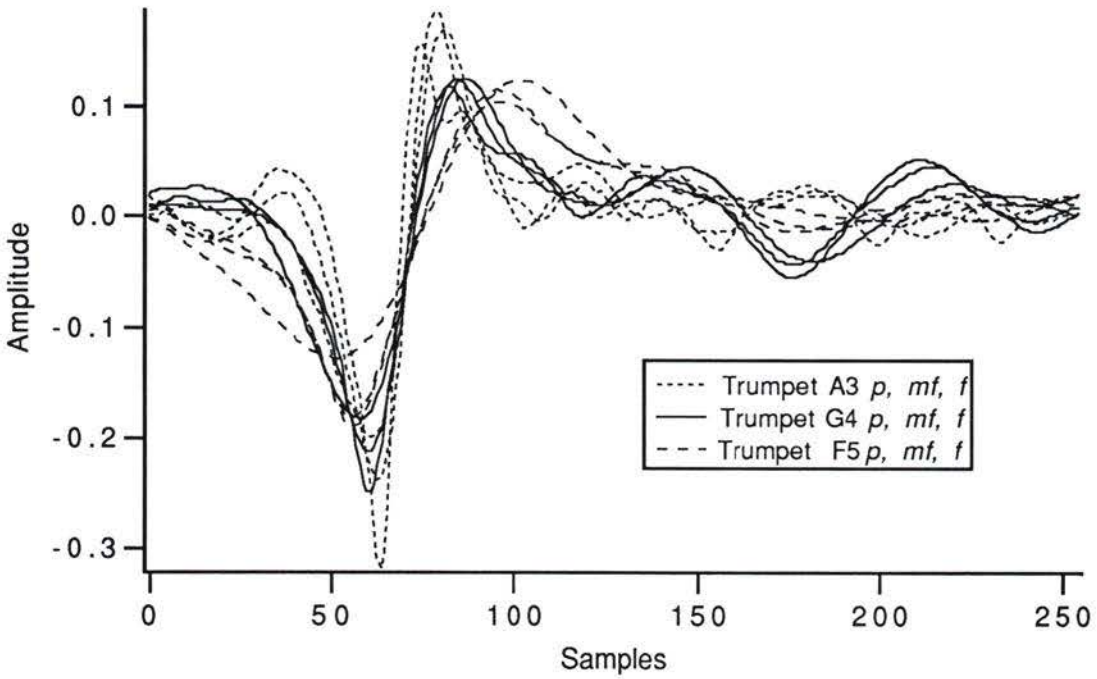

Figure 1.1 - Single-Period Segments of Trumpet Tones played at Different Pitches and Dynamic Levels

Given that multiple samples have been recorded, for any given playback note, the closest pre-recorded sample in terms of pitch and volume is appropriately scaled for sample-playback. Pre-recorded samples are most effective when they are fed through some kind of post-processing unit on playback. Useful post-processing includes filtering, equalization and artificial reverberation. Many synthesizers include this post-processing capability.

One drawback to sample-playback synthesis is the amount of memory typically required to store the many pre-recorded samples. One solution to this drawback is wavetable synthesis. Wavetable synthesis works on the

assumption that a musical tone is nearly periodic. A single period of a waveform is simply repeated, using a wavetable oscillator, for the duration of the tone. The single-period segments of trumpet tones shown in Figure 1.1 could be used as wavetables for synthesizing trumpet-like tones. Alternatively, arbitrary wavetables could be defined. The drawback of wavetable synthesis is that a strictly periodic tone quickly becomes tedious and mechanical. Wavetable synthesis is typically augmented with vibrato, tremolo and amplitude envelope control on the wavetable oscillator.

A problem with both sample-playback synthesis and wavetable synthesis is providing the performer with expressive control over significant tone-varying parameters. A variation on wavetable synthesis which attempts to provide more expressive control is vector synthesis. Vector synthesis is essentially wavetable synthesis using multiple wavetables. These wavetables can be played simultaneously or chained. Independent frequency and amplitude controls are applied to each wavetable. The amplitude level of each wavetable can be thought of as an independent dimension of control. The recently introduced line of Korg Wavestations implement vector synthesis. Synthesis based on the Karhunen-Loève Transform (KLT), the topic of this thesis, is really just a constrained form of vector synthesis, where an analysis algorithm incorporating the KLT derives the wavetables.

Synthesis based on the wavelet transform can also be viewed as a form of processed recording. A sampled tone is analyzed to produce a time-dilation map of wavelets. Synthesis is achieved by playing back wavelets according to the map. Kronland-Martinet has applied the wavelet transform to analyzing and re-synthesizing musical tones [11]. Perhaps the wavelet transform could be termed highly-processed recording.

Spectral Modeling

Some spectral modeling techniques can also be viewed as processed recording; however they can all be distinguished from the processed recording techniques given above by their emphasis on frequency-domain signal representation vs. time-domain signal representation. One advantage of spectral modeling techniques is that working with a tone's frequency content is closely related to the human hearing mechanism. It is well-accepted that the human ear and auditory processing system performs some level of frequency analysis [12]. Spectral modeling can be viewed as placing an emphasis on the *receiver* of musical tones.

Perhaps the most readily understood spectral modeling synthesis technique is additive synthesis. Most generally, additive synthesis is based on adding up several simpler waveforms to produce a complex waveform. In that sense, the vector synthesis technique mentioned earlier can be considered a type of additive synthesis. But additive synthesis traditionally refers to the process of adding up multiple sinusoidal oscillators to produce a more complex signal. In the most traditional additive synthesis, these oscillators are harmonically related. Musical tones can be processed using Fourier analysis to determine the relative amplitudes of each of the oscillators. Additive synthesis has been the focus of much research effort, and has thus been refined significantly. Serra and Smith have developed algorithms for automatically analyzing and re-synthesizing non-harmonic as well as harmonic tones [13], and characterizing the noise-like or stochastic component of tones for re-synthesis [14]. The great problem with additive synthesis and its refinements is the amount of computation required in re-synthesis, with easily 30 oscillators running for one tone. A competing

approach to additive synthesis, where multiple distinct oscillators are running in the time-domain, is to simply retain frequency domain information and perform successive Inverse FFT's for synthesis. Recent work on this approach has been performed by George and Smith [15].

Subtractive synthesis, while the complement of additive synthesis, can be viewed as a spectral modeling technique. In subtractive synthesis, a harmonically rich signal, such as a pulse train, is fed through filters to shape the spectrum. Subtractive synthesis was the typical technique used in analog synthesizers and electronic organs for many years. Implementation of subtractive synthesis with digital hardware was performed by Goeddel and Bass [16,17]. One of the problems which they resolved was choosing a discrete-time excitation signal that was harmonically rich but did not result in aliasing. They also developed an automatic analysis/re-synthesis procedure.

Physical Modeling

Unlike spectral modeling, which emphasizes the receiver, physical modeling focuses on the *transmitter* of musical tones. As the category name implies, these synthesis algorithms attempt to model the physical acoustics of an existing acoustic instrument. Of course, the parameters of any model could be modified to emulate new, non-existent instruments. A good physical model should be able to emulate different playing techniques over the full range of the instrument. Physical models provide a direct connection between performance controls and the synthesis algorithm. One drawback of physical models is that a different algorithm is needed for each acoustic instrument to be emulated. Another drawback can be in the computational power required to execute a detailed model.

Work on synthesis using physical models is still in its early stages. While not the exclusive source of research in this area, the Stanford Center for Computer Research in Music and Acoustics has made significant contributions to physical modeling synthesis. Three recent papers by Cook, Hirschman and Smith [18,19,20] indicate something of the current state of research in this area.

Abstract Algorithms

The most significant abstract algorithm, at least in terms of commercial synthesizers, is frequency modulation (FM) synthesis. The famous Yamaha DX-7 line of synthesizers employ this technique first developed by Chowning [21]. FM synthesis affords complex time-varying spectra with a small number of sinusoidal oscillators. A surprisingly large number of acoustic instrument tones have been well-approximated using FM synthesis. FM works particularly well on bright percussive and metallic sounds. The one common complaint about FM is the "metallic edge" on almost any synthesized tone. Another problem with FM synthesis is the lack of a general, automated analysis technique to generate the synthesis parameters. It is not possible to feed sampled tones into an FM analysis "black box" and get the synthesis parameters required for re-synthesis at the output. Some work has been done in this area by Payne [22], but it remains a difficult and largely unsolved problem. More recent implementations of FM synthesizers have gone beyond just using sine wave oscillators to using square waves, triangle waves, sawtooth waves or even arbitrary multiple-period wavetables.

1.2. Issues and Objectives in Musical Tone Analysis/Synthesis

The issues which arise in musical tone analysis/synthesis involve a knowledge of the fundamental parameters or features of musical tones. Chamberlin provides a reasonable description of musical tone parameters as well as synthesis goals in his book [23]. The following two sections elaborate on this material.

1.2.1. Features of Musical Tones

There are a number of features of musical tones which need to be considered in developing an analysis/synthesis scheme. Any musical tone is an event which evolves over time. An important measure of a tone's evolution over time is its amplitude envelope. Figure 1.2 shows the amplitude envelope of a typical trumpet tone.

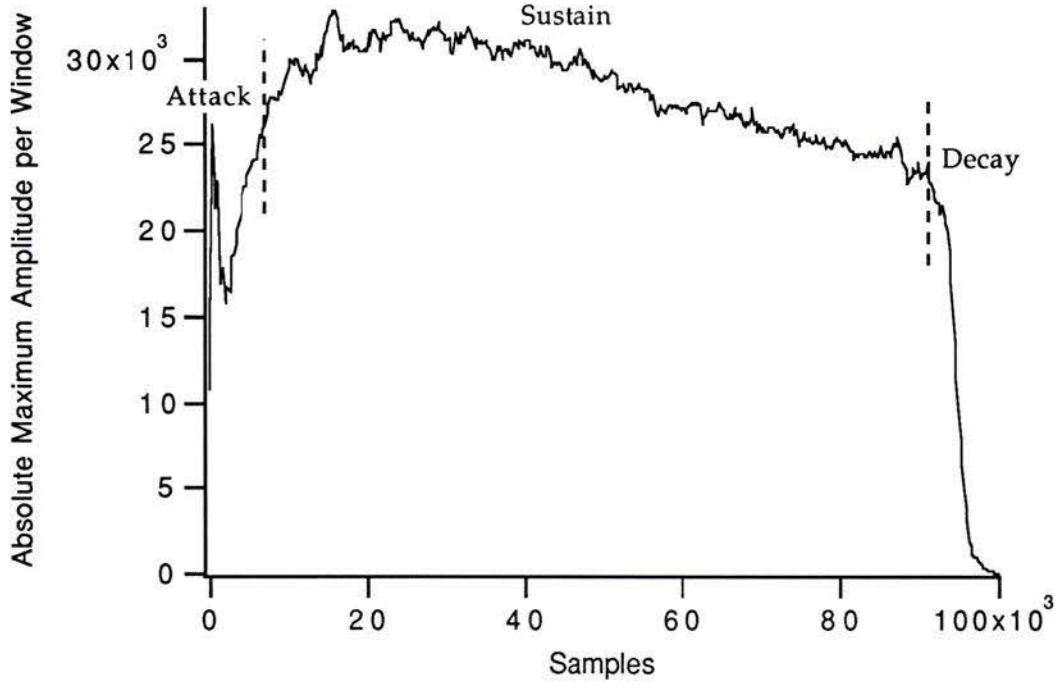

Figure 1.2 - Amplitude Envelope of a Trumpet Tone

The envelope is derived by stepping through the tone window by window and choosing the maximum absolute sample value in each window (similar to a method described in [24]). For a near-periodic tone such as the trumpet, the window should be at least one period long. If the window is too long, then obviously some time resolution is lost.

The amplitude envelope of Figure 1.2 is roughly divided into three regions: attack, sustain and decay. The sustain region is the portion of the tone which can be best termed near-periodic. During the attack, the tone commences and evolves rapidly in amplitude and timbre (frequency content) to the somewhat steady-state amplitude and timbre of the sustain region. During the decay, the tone amplitude fades away to zero, again with the tone's timbre changing as it dies away. The amplitude envelope only shows the amplitude variation during the attack and decay. Some of the variation in the early attack and late decay can be seen by looking at the time domain signal, as given in Figures 1.3 and 1.4.

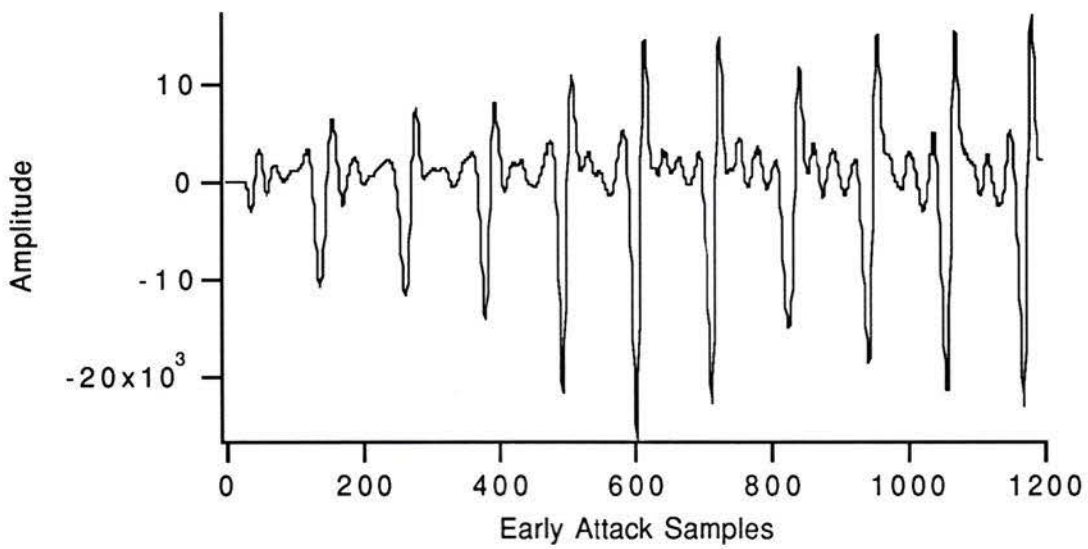

Figure 1.3 - Early Attack of a Trumpet Tone

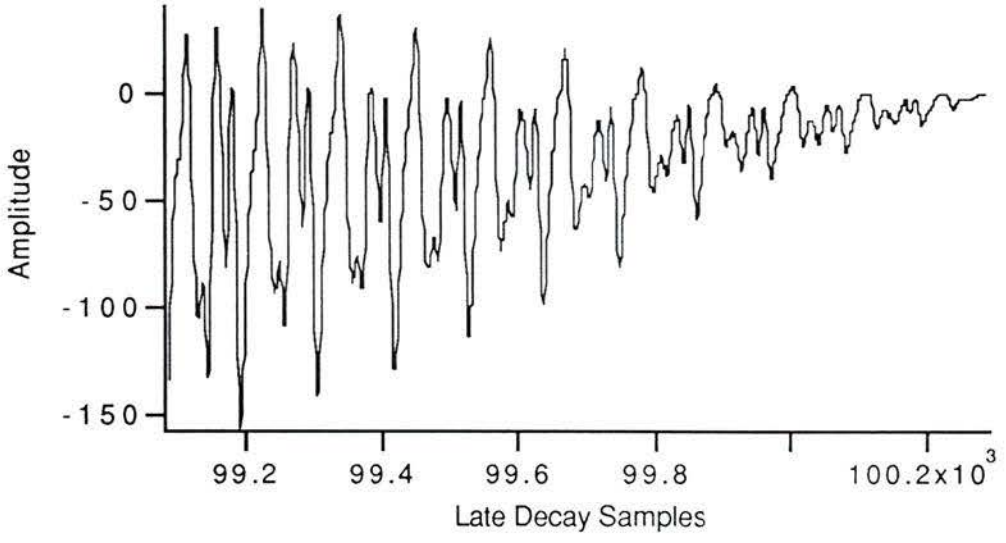

Figure 1.4 - Late Decay of a Trumpet Tone

It is easier to see the attack and decay timbral variation by viewing the frequency spectrum of the trumpet tone over time, as given in Figure 1.5.

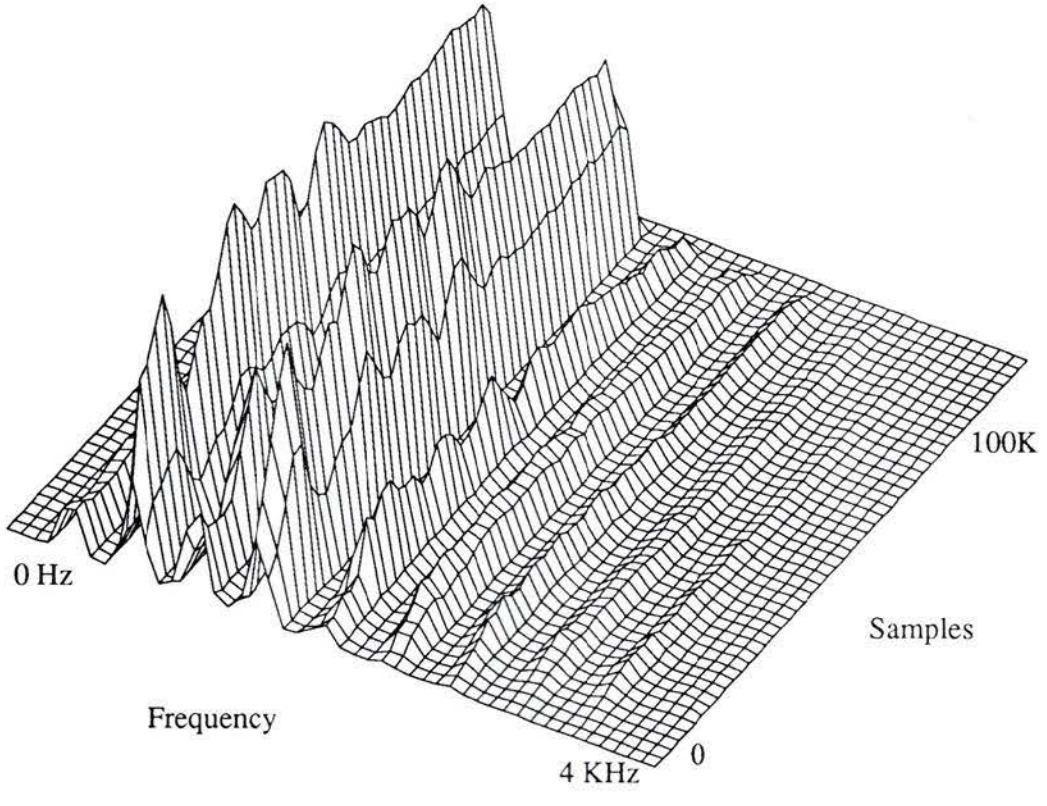

Figure 1.5 - Time-Varying Spectrum of a Trumpet Tone

During the attack and decay, different harmonics grow and fade at differing rates. The higher frequency harmonics grow at a slower rate in the attack and fade away sooner in the decay. Attack, sustain and decay features vary from instrument to instrument, and even for one instrument's tones depending on the playing technique. Both attacks and decays may be long or short. Percussive instruments really don't have a sustain region. They typically have rapid attacks followed by decays with short to long durations.

Two other common time-varying features of musical tones are vibrato and tremolo. Vibrato is near-periodic pitch variation of a tone, and tremolo is near-periodic amplitude variation of a tone. For many instruments, these are aspects of a tone which can be controlled by the performer - adding none, some or a lot of either feature while playing. For example, a trombone player achieves vibrato by rapidly moving the slide in and out over a short range. The player has control over both the rate and depth (amount of pitch variation) of vibrato.

An important frequency domain feature of musical tones is whether a tone is harmonic or non-harmonic. A harmonic tone's frequency spectrum has peaks which are located at or near integer multiples of the tone's fundamental frequency. Wind and string instruments are typically harmonic. Conversely, a non-harmonic tone has significant peaks which are not integer multiples of the tone's fundamental frequency. Bells and mallet instruments such as the marimba and xylophone fall into the category of non-harmonic tones.

No acoustic instrument tone can be viewed strictly as the sum of time-varying sinusoidal components. There is also a noise-like or stochastic component to tones. Typically, to achieve realistic synthesis of such tones, it is

important to model and reproduce an approximation of the stochastic component.

Finally, in summing up all these features, there is no one tone which adequately represents all of the tones playable on a given instrument, as mentioned earlier in the discussion of Processed Recording synthesis. Thus the tone features for any given instrument must be taken from several different playing styles, dynamic levels and pitches to get a true representation of the instrument's features.

1.2.2. Digital Synthesis Algorithm Benchmarks

There are several benchmarks, or measuring sticks, which can be used in comparing digital synthesis algorithms. They include the quality of synthesized tones, the range of tones which can be synthesized, the existence of an analysis algorithm and computation/memory requirements.

The first measure of a synthesis algorithm is the quality of tones it can produce. Are the tones "rich enough?" This measure can be rather subjective. A more measurable quality benchmark typically used is the ability of the synthesis algorithm to realistically reproduce the tones of an acoustic instrument. There are two reasons for using this benchmark. First, the tones of acoustic instruments are typically considered "pleasing" and "rich." If the synthesis algorithm can adequately reproduce acoustic instrument tones, then it should also be able to generate new and different tones of quality equal to acoustic instrument tones. Second, it is possible to conduct formal comparative listening tests between the original and synthesized tones.

A synthesis algorithm can be restricted in terms of the range of tone types or features which it can produce. Wavetable synthesis is, in its simplest form,

restricted to synthesizing harmonic tones. One algorithm may do a much better job of synthesizing tone attacks than another algorithm.

The usefulness of a synthesis algorithm can also be measured by whether or not there is a corresponding analysis algorithm. For any given synthesis algorithm, is it possible to process a sampled tone and generate the necessary parameters for imitating that tone with the algorithm? Frequency Modulation synthesis, while known for the rich range of tones it can produce, is limited by the absence of a thorough analysis algorithm, as mentioned previously.

The final benchmark for a synthesis algorithm relates to implementation cost - the computational and memory requirements for generating a tone or range of tones. Computational cost can usually be reduced to considering the number of multiplies and additions required to produce a single output sample for a tone. Memory cost can be quantified as the amount of storage required for all the parameters and waveform data required to reproduce the various tones of one instrument. Typically there exists a tradeoff between computational cost and memory cost when comparing different algorithms.

1.3. The Karhunen-Loève Transform

The application of the Karhunen-Loève Transform (KLT) to musical tone analysis/synthesis is relatively new. The KLT has traditionally been applied in the problem of signal compression. Some specific applications have been in the areas of digital image compression [25], seismic data compression [26] and electrocardiogram compression [27]. The KLT is applied by representing segments of discrete signal data as random vectors or sample functions in a random process. The function of the KLT is to orthogonalize the space of a

random process. Applying the KLT to a set of sample functions representative of the random process yields a set of orthonormal vectors, called basis vectors or functions. The original sample functions can then be expressed as a weighted sum of the basis functions [28]. The important property of these KL basis functions is that the form of a given sample function is concentrated in the first few basis functions if the sample functions are reasonably correlated. Thus, most of the basis functions can be discarded, and each sample function can be approximated with a minimum of error by a small set of common basis functions and its own weights for summing the basis functions. Also, sample functions similar but not identical to the original sample functions from which the KL basis functions were derived, can be well-approximated by the truncated set.

1.3.1. Definition of the KLT

Consider a discrete random process represented by sample functions $x_i(k)$, $k = 1,2,\ldots,K$. Each sample function can be represented as a K-by-1 random vector $\mathbf{x}_i$, the *kth* sample of $x_i(k)$ corresponding to the *kth* element of $\mathbf{x}_i$. It is useful to expand each sample function $\mathbf{x}_i$ (hereafter $\mathbf{x}_i$ and $x_i(k)$ will be used interchangeably to denote a sample function) as the weighted sum of K orthonormal K-by-1 basis vectors (or functions) spanning K-space.

$$\mathbf{x}_i = \sum_{n=1}^{K} w_{i,n}\boldsymbol{\beta}_n \tag{1-1}$$

where $w_{i,n}$ is the weight of the *nth* basis function $\boldsymbol{\beta}_n$. To be orthonormal the basis functions must satisfy the following requirement.

$$\boldsymbol{\beta}_p^T\boldsymbol{\beta}_q = \begin{cases} 1, & p = q \\ 0, & p \neq q \end{cases} \tag{1-2}$$

If the sample functions in a random process are reasonably correlated, it is possible to reasonably approximate a sample function with less than K basis functions as follows.

$$\mathbf{x}_i \approx \sum_{n=1}^{N} w_{i,n}\boldsymbol{\beta}_n, \qquad N \ll K \tag{1-3}$$

In this case, the problem becomes one of choosing an orthonormal set of basis functions such that the first N are the best functions for approximating the sample functions. The KLT satisfies the above loose criteria with the following approximation-error-minimizing properties:

1) minimum mean square truncation error,
2) uncorrelated transform coefficients, and
3) minimum entropy expansion for a random process.

The KLT basis functions are derived as follows. First the random process correlation matrix $\mathbf{R}_\mathbf{x}$ is estimated. The correlation matrix is defined as the expected outer product of a sample function in the process.

$$\mathbf{R}_\mathbf{x} = \mathrm{E}\left\{\mathbf{x}_i\mathbf{x}_i^T\right\} \tag{1-4}$$

If a finite number M of independent sample functions are taken as representative of the process, then $\mathbf{R}_\mathbf{x}$ can be estimated as

$$\mathbf{R}_\mathbf{x} \approx \frac{1}{M}\sum_{i=1}^{M}\mathbf{x}_i\mathbf{x}_i^T. \tag{1-5}$$

For $\mathbf{R}_\mathbf{x}$ to be full rank, M must be greater than or equal to the sample function length K.

The second step is to compute the eigenspace of $\mathbf{R}_\mathbf{x}$. The eigenspace of $\mathbf{R}_\mathbf{x}$ is characterized as

$$\mathbf{R}_X \boldsymbol{\beta}_n = \lambda_n \boldsymbol{\beta}_n, \qquad n = 1,2,\ldots,K \tag{1-6}$$

where λ_n, a scalar, is an eigenvalue of $\mathbf{R}_X$, and $\boldsymbol{\beta}_n$ is the K-by-1 eigenvector associated with λ_n. The eigenvalues are ordered such that

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0. \tag{1-7}$$

Each of the $\mathbf{x}_i$ contributing to the estimate of $\mathbf{R}_X$ given in (1-5) can be exactly computed using (1-1). The weights for each sample function $\mathbf{x}_i$ are

$$w_{i,n} = \mathbf{x}_i^T \boldsymbol{\beta}_n, \quad i = 1,2,\ldots,M,\ n = 1,2,\ldots,K. \tag{1-8}$$

where the $\boldsymbol{\beta}_n$ are taken from (1-6). Using only the first N ($N<K$) basis functions from (1-6) and weights from (1-8), each of the $\mathbf{x}_i$ can be approximated with minimum mean-squared error for a rank-N approximation using (1-3).

1.3.2. Properties of the KLT

The error-minimizing properties of the KLT are presented thoroughly with derivations in [29]. Here we will simply describe the properties. The property of minimum mean square truncation error is defined as follows. First, compute a sample function $\mathbf{x}_i$ using (1-1).

$$\mathbf{x}_i = \sum_{n=1}^{K} w_{i,n} \boldsymbol{\beta}_n$$

Then compute an approximation to the sample function $\tilde{\mathbf{x}}_i$ using (1-3).

$$\tilde{\mathbf{x}}_i = \sum_{n=1}^{N} w_{i,n} \boldsymbol{\beta}_n, \qquad N < K \tag{1-9}$$

The error due to truncation is $\Delta\mathbf{x}_i = \mathbf{x}_i - \tilde{\mathbf{x}}_i$. The mean square truncation error is defined to be

$$\varepsilon = \mathrm{E}\left\{\Delta\mathbf{x}_i^T \Delta\mathbf{x}_i\right\}. \tag{1-10}$$

Obviously, we want ε to be as small as possible for a given value of N in (1-9). It can be shown that by choosing basis functions which are the eigenvectors of the correlation matrix $\mathbf{R}_x$, and ordering them as implied by (1-7), ε is minimized. The value of ε for this case is

$$\varepsilon = \sum_{n=N+1}^{K} \lambda_n . \tag{1-11}$$

The transform coefficients are simply the weights $w_{i,n}$ defined in (1-8). If the KLT is used to compute the basis functions, these weights are uncorrelated, that is

$$\mathrm{E}\{w_{i,p} w_{i,q}\} = \begin{cases} \lambda_p = \lambda_q, & p = q \\ 0, & p \neq q \end{cases} \tag{1-12}$$

This property is of importance in showing that the KLT is the minimum entropy expansion for a Gaussian random process (considering only second order statistics).

Entropy is a statistical measure of uncertainty [30]. The expansion of (1-1) is termed to have minimum entropy when the entropy of the basis functions representing the process is minimized. The entropy of a basis function set corresponds to uncertainty about the relative importance of individual basis functions in the set. High entropy indicates that it is not possible to efficiently truncate the representation of the process to just a few of the basis functions. Low entropy indicates that there are just a few basis functions which represent most of the energy in the process. The entropy of the basis functions used to represent a random process is

$$\mathrm{S}(\{\boldsymbol{\beta}_n\}) = -\sum_{n=1}^{K} \rho_n \log \rho_n . \tag{1-13}$$

where ρ_n is the normalized average process energy represented by the basis function $\boldsymbol{\beta}_n$. (The ρ_n are normalized such that $\sum_{n=1}^{K}\rho_n = 1$. Note also that $\rho_n \geq 0$ for all n.) The higher ρ_n, the more important $\boldsymbol{\beta}_n$ is in representing a sample function in the process. For any given random process, (1-13) is minimized if $\boldsymbol{\beta}_n$ are the eigenvectors of the process correlation matrix $\mathbf{R}_x$. Furthermore, the energies of each basis function, ρ_n, are just the eigenvalues, λ_n. Hence, (1-13) can be rewritten as

$$S(\{\boldsymbol{\beta}_n\}) = -\sum_{n=1}^{K}\lambda_n \log \lambda_n . \tag{1-14}$$

1.4. Synthesis based on the Karhunen-Loève Transform

Synthesis based on the Karhunen-Loève Transform, as described previously in the section on Processed Recording, is really just vector synthesis distinguished by the tone analysis algorithm or procedure. In order to apply the KLT to analyzing musical tones, it is necessary to represent musical tones as a random process. A set of discrete-time sample functions must be extracted from one or more musical tones which in some way are representative of the tones. These sample functions must all be of the same length K. In the original work of Stapleton and Bass [29,31], the representative sample functions were chosen to be single-period segments from one or more tones. The rationale for choosing single-period segments is based on assuming the tones are periodic. No acoustic instrument tone is strictly periodic, but many are near-periodic. For such instruments, if several single-period segments are extracted from across the duration of one tone, a reasonable representation of that tone is obtained. Figure 1.6 shows a single-

period segment extracted from a trumpet tone. We can see that the tone is nearly, though not perfectly, periodic over the three cycles shown.

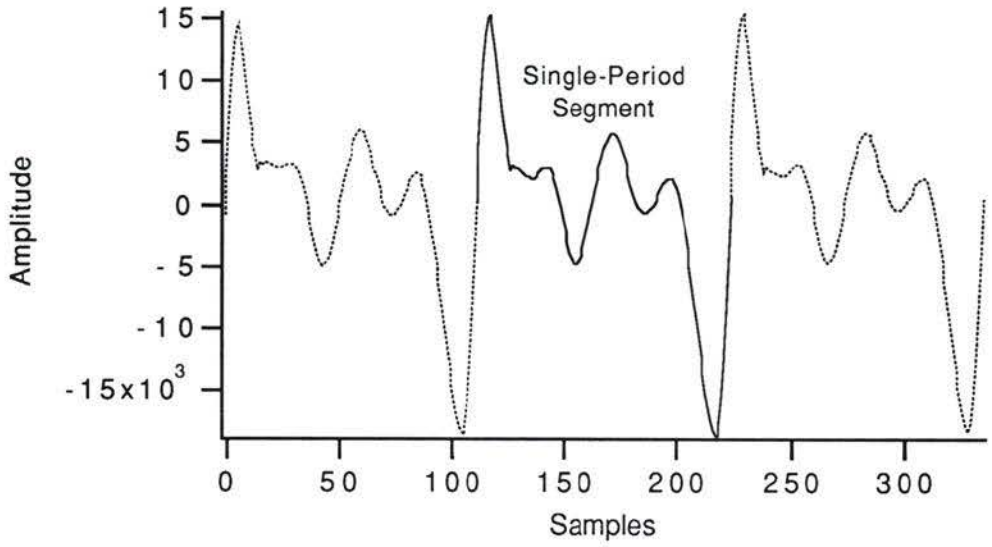

Figure 1.6 - Single-Period Sample Function Extracted from a Trumpet Tone

The sample period of this trumpet tone is between 112 and 113 samples. The tone was played at a pitch (frequency) of approximately 392 Hz, corresponding to the note G4, and sampled at a rate of 44.1 KHz. The sample period of any other tone will vary directly with the rate at which it is sampled, and inversely with the tone's pitch. Sample functions taken from several tones will typically not have the same sample period. Thus, a sample period normalization must be performed in extracting the sample functions. This can be done with a sample rate conversion filter.

Having extracted single-period segments from one or more tones, basis functions are computed using the KLT. Tones can be re-synthesized from a truncated set of the first N basis functions as follows:

$$s(k) = \sum_{n=1}^{N} A_n(k)\beta_n(\phi(k)) \qquad (1\text{-}15)$$

The basis functions β_n are periodic extensions for the length K basis vectors computed with the KLT. The argument of the basis functions $\phi(k)$ is a time-varying phase index. All basis functions share the same phase index. This is required to ensure that the basis functions remain orthogonal. This also restricts re-synthesis to harmonic or near-periodic tones, since the cycle rate of the individual basis functions cannot be independent. Each basis function does have its own unique time-varying amplitude coefficient $A_n(k)$. The time-varying amplitude coefficients and phase index can be automatically determined by analyzing a tone suitable for re-synthesis by the basis functions (for example, one of the tones from which the original single-period sample functions were taken).

1.5. Prior Work on KL-based Synthesis

While KL-based synthesis is best viewed as vector synthesis with an optimal analysis procedure, Stapleton and Bass originally developed the technique as an alternative approach to Fourier-based additive synthesis. KL-based synthesis was developed in an attempt to provide an alternative additive synthesis technique with significantly reduced computational load when compared with Fourier-based additive synthesis. The high computational load of Fourier-based additive synthesis is due to the large number of oscillators typically required. KL-based synthesis reduces the number of oscillators required by using oscillator waveforms which are more complex then simple sine waves. Typically, the number of oscillators can be

reduced from thirty or more to three or four. This yields as much as an order of magnitude reduction in computational cost.

The reduction in the number of oscillators is traded off against three factors: possible increased memory, risk of aliasing and loss of generality. Increased memory is required for the more complex oscillator waveforms; however, less memory is required for oscillator control data (for example, amplitude envelopes) since there are fewer oscillators. It is possible that KL-based synthesis could actually require less memory if highly detailed oscillator control data is used.

KL-based synthesis introduces a need for dealing with aliasing in a different way than Fourier-based synthesis. With Fourier-based synthesis of high-pitched notes, aliasing can be prevented by simply not running sinusoidal oscillators at frequencies which exceed half the sampling rate. With KL-based synthesis the basis functions are harmonically rich, each having the same bandwidth as the normalized single-period sample functions. If these basis functions are used to synthesize high-pitched notes, aliasing can easily occur. Thus basis functions must be band-limited such that they will not alias for the highest pitch which will be synthesized. When the same basis functions are used for the full frequency range of an instrument, band-limiting required for high-pitched notes forces a decrease in the harmonic richness of low-pitched notes.

The final significant trade off is the loss of generality, which manifests itself in two ways. The first loss of generality is that it is not possible to find a small number of oscillator waveforms which can be used to represent a broad cross-section of acoustic instruments. The sine wave oscillators of Fourier-based additive synthesis can be used to represent almost any instrument. KL-

based synthesis requires that different oscillator waveforms be derived for different instruments or groups of similar instruments. The second loss of generality is in the range of tones which can readily be synthesized. Fourier-based synthesis has been developed to handle both harmonic and non-harmonic tones; KL-based synthesis, as developed by Stapleton and Bass, only handles harmonic (near-periodic) tones.

Besides the above comparisons, Stapleton and Bass compared KL-based synthesis with Fourier-based synthesis in the context of implementing a voice chip for an electronic organ. Their conclusions were that a Fourier-based voice chip would typically require three times the number of transistors as a KL-based voice chip, making the KL-based voice chip more feasible and cost-effective.

In originating KL-based musical tone synthesis, Stapleton and Bass developed the analysis procedure for automatically analyzing an input tone to produce the basis function oscillators and required synthesis parameters. They applied the analysis procedure to fourteen acoustic instruments, using only one tone per instrument. These tones were grouped using a classification procedure which measured the correlation between tones. The classification procedure divided the fourteen tones into three highly correlated groups and two tones which did not fit well into any group. The two unclassified tones were successfully added to the group with which they correlated the best. The KL basis functions for each of these three groups were computed, and tones were re-synthesized from the basis functions. The re-synthesized tones were judged to be realistic representations of the original tones in informal listening tests. An alternative approach to classification was also tried. All fourteen tones were grouped together for derivation of a set of

common KL basis functions. Here, the lack of generality of KL basis functions became evident, as ten or more basis functions were typically required for synthesis of any one tone.

1.6. Contributions of this Thesis

The first contribution of this thesis is to verify the work of Stapleton and Bass. The analysis procedure they developed has been implemented in *Matlab* on a Sun workstation. Some minor variations have been made to the procedure they described. First, multiple estimates are made of a tone's sample period to provide a reliable averaged estimate. Second, the sample rate conversion filter used to normalize the sample period of single-period sample functions has been implemented in a more efficient manner than they described. Third, some simplifications and extensions have been made to the procedure for extracting the amplitude envelopes and phase function used in re-synthesizing a tone from its basis functions. Overall, an attempt has been made to further formalize a number of steps in the original procedure, presenting more of a "cookbook" approach. These contributions are detailed in Chapter 2.

The second contribution of this thesis is to use the KL analysis procedure for further analysis of instrument tones. In particular the focus of our work in analyzing tones has been to restrict a set of KL basis functions to one instrument. First the KL basis functions have been derived for one single tone of a given instrument. Then a number of tones from across the pitch and dynamic ranges of an instrument have been grouped together to derive a set of common KL basis functions. This allows us to investigate how well a small set of KL basis functions can track the timbral variations across the

playing range of an instrument. Finally tones of different playing techniques (plucked vs. bowed string) for a single instrument have been grouped together for analysis. Chapter 3 presents quantitative and qualitative results of our analysis/synthesis experiments.

The final contribution of this thesis is to suggest improvements or extensions which could be made to the KL analysis procedure. None of these extensions are novel in the context of all analysis/synthesis algorithms. The extensions suggested have all been developed to improve the quality and flexibility of other algorithms, but have not been implemented with KL-based analysis/synthesis. A description of these extensions and how they could be incorporated into the KL analysis/synthesis procedure is presented in Chapter 4 as suggestions for future work.

An intended contribution of this thesis was to implement a real-time prototype of a KL-based synthesizer. Having a real-time KL-based synthesizer available would allow quicker auditioning of analysis/synthesis results. Preliminary work has been done on this but it was decided that the time required to complete the work was prohibitive. Regardless of time considerations, the value of such work is reduced by the introduction of synthesizers such as the Korg Wavestation which implements the necessary components of KL-based synthesis.

Chapter 5 presents a summary and conclusions of the thesis.

CHAPTER 2
THE KARHUNEN-LOEVE ANALYSIS/SYNTHESIS PROCEDURE

In this chapter we present a detailed description of the KL-based musical tone analysis/synthesis procedure. The software to implement the procedure is briefly described in the final section. The KL-based musical tone analysis procedure consists of 3 major steps, as shown in Figure 2.1.

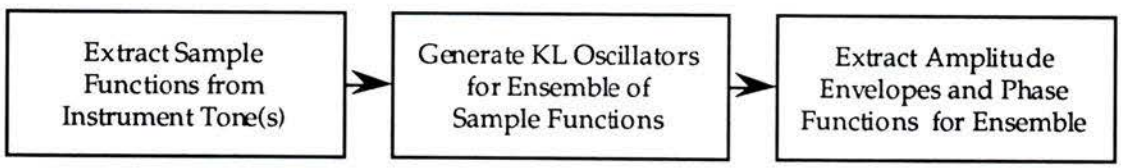

Figure 2.1 - KL Analysis Procedure

The first step of the procedure performs preliminary analysis and processing of a sampled instrument tone in order to extract a number of single-period sample functions, normalized in sample period and energy. The second step of the procedure groups sample functions from one or more instrument tones (all normalized to the same sample period and energy), phase-aligns the sample functions, and finally performs the KLT on the sample functions to produce KL basis functions. The final step of the procedure analyzes each instrument tone contributing to the KL basis function set to determine the amplitude envelopes and phase functions required to synthesize each tone from the KL basis functions. For the most part this procedure follows the one introduced by Stapleton and Bass [29,31].

2.1. Preliminary Tone Analysis and Sample Function Extraction

In extracting sample functions from musical tones, we choose to restrict the range of extraction to the sustain region. It is over the sustain region of a harmonic instrument's tone that the waveform is near-periodic. The preliminary tone analysis consists of identifying a tone's sustain region and determining the tone's sample period in the sustain region. The sample function extraction first consists of extracting several short, multi-period buffers from a tone's sustain region and passing these buffers through a sample rate conversion filter to normalize the sample period. Second, a single period sample function is extracted from each re-sampled buffer using a scheme to ensure the sample function is periodically extensible. Finally, each sample function is normalized to unit energy.

2.1.1. Preliminary Tone Analysis

The first step in the preliminary tone analysis is to determine a tone's sustain region. We have chosen to implement this step interactively with a human user, versus performing some kind of automatic pattern recognition scheme. The amplitude envelope of the tone being analyzed is presented graphically to a user for selecting the bounds of the sustain region, as shown in Figure 2.2. The amplitude envelope is determined following the scheme described in Chapter 1, Section 1.2.1. A window of length two sample periods is stepped through the tone without overlapping. For each two-period window, the maximum absolute value of the tone's waveform is retained as the amplitude envelope value. The choice of two periods for the window length is somewhat arbitrary. A window length less than one period is too short for tracking long-term tone amplitude variation. Too long a window

length sacrifices time resolution. In this case, where the bounds of the sustain region are just being roughly chosen, window lengths several times longer than two periods would probably be acceptable.

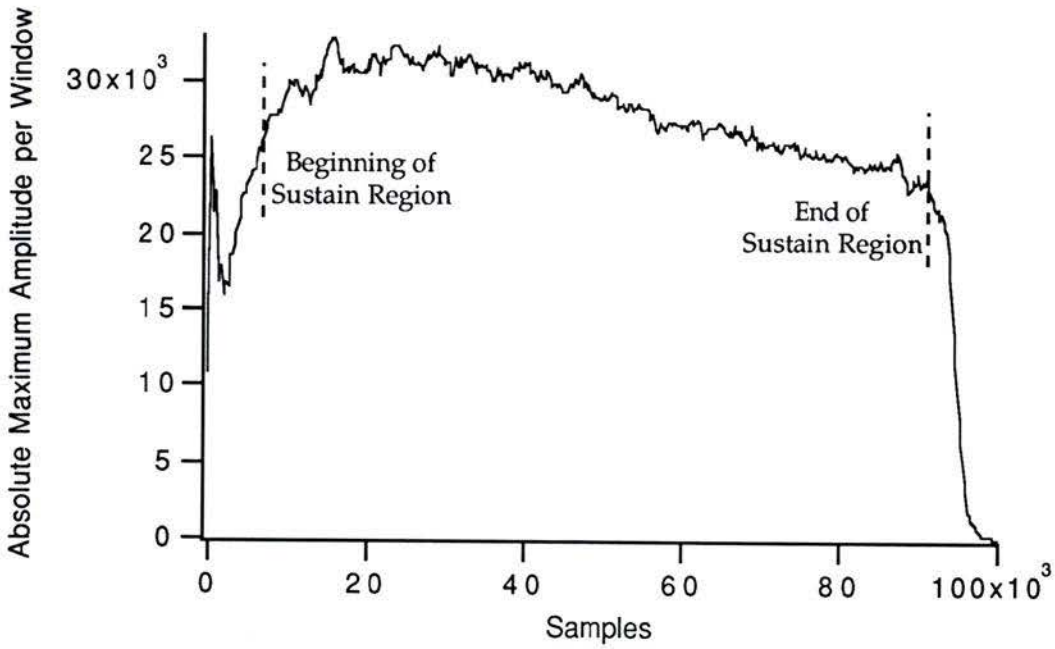

Figure 2.2 - Sustain Region selected from a Trumpet Tone

A reasonable estimate of a tone's sample period, which is used in determining the amplitude envelope window length, is known a priori based on a knowledge of the sampling rate at which the tone was recorded and the actual note played. A commonly used text notation for notes is shown in the first column of Table 2.1. The notes extend over one octave from C4 (middle C) through C5. The frequencies given for each note are based on an equal-tempered scale. These frequencies can be calculated as

$$f_{note} = 2^{(12N_O+N_N)/12} f_{C0} \tag{2-1}$$

where N_O is the note's octave number (ranging from 0 through 8 to include all notes on a piano), N_N is the note's number (ranging from 0 through 11),

and f_{C0} is the reference frequency of the note C0, approximately 16.352 Hz. This note falls below the low-frequency threshold of 20 Hz for human hearing. The lowest note on a piano, A0, has a frequency of precisely 27.5 Hz, from which the frequency of C0 is derived as

$$f_{C0} = 2^{-3/4} f_{A0}. \tag{2-2}$$

The sample period K_{note} (the number of samples in one period) of the tone sample, expressed as a dimensionless integer, is the rounded value of the sampling frequency f_s divided by the note frequency f_{note}.

$$K_{note} = \text{round}(f_s / f_{note}) \tag{2-3}$$

Note [text]	Note Number	Octave Number	Frequency [Hz]	Sample Period at f_S=44.1KHz
C4	0	4	261.624	169
C#4	1	4	277.200	159
D4	2	4	293.656	150
D#4	3	4	311.124	142
E4	4	4	329.648	134
F4	5	4	349.228	126
F#4	6	4	370.040	119
G4	7	4	392.040	113
G#4	8	4	415.316	106
A4	9	4	440.000	100
A#4	10	4	466.136	95
B4	11	4	493.856	89
C5	0	5	523.248	84

Table 2.1 - Standard Note Parameters for a One Octave Range of the Equal-Tempered Scale

Even though an estimate of a tone's sample period is available a priori, we choose to verify and possibly adjust this value by analyzing the tone's sustain region to extract a computed estimate. The procedure for computing an estimate of the sample period is as follows.

1) Extract several multi-period buffers $y_i(k)$ ($i = 1 \ldots N$ for N buffers) evenly distributed across the tone's sustain region. (Note that k is a discrete-time sample index. For each buffer it ranges over several sample periods.)

For each buffer $y_i(k)$ estimate the tone's sample period $K_{tone,i}$ as follows.

2) Find the first zero-crossing in the buffer at k_0. Note that the sample indexed by k_0 is typically not the exact zero-crossing location (which is not always represented in the discrete-time sampled data), but the sample either immediately before or after a zero-crossing. (The choice to use either the sample before or after a zero-crossing should be made consistently throughout this procedure.)
3) Using the a priori sample period estimate K_{note}, find all zero crossing sample indices k_j of the same slope in the range $k_0 + \frac{1}{2}K_{note}$ through $k_0 + \frac{3}{2}K_{note}$.
4) Compute the correlations c_j between a single sample period starting at k_0 and single sample periods starting at each of the k_j.

$$c_j = \sum_{k=0}^{K_{note}-1} y_i(k_0 + k) y_i(k_j + k) \tag{2-4}$$

5) The zero crossing at k_j corresponding to the largest correlation c_j is used to compute the sample period for the buffer.

$$K_{tone,i} = k_j - k_0 \tag{2-5}$$

6) Having computed $K_{tone,i}$ for each buffer, the average sample period from all the buffers is then computed as

$$K_{tone} = \text{mean}\left(K_{tone,i}\right) \tag{2-6}$$

or

$$K_{tone} = \text{median}\left(K_{tone,i}\right). \tag{2-7}$$

Equation (2-7) is preferred to eliminate the effect of any isolated wild estimates of the tone period.

The computed estimate K_{tone} is used in place of the a priori estimate K_{note} as the tone's sample period. Note that we restrict the range of permissible values for K_{tone} to be integer.

2.1.2. Extracting Period-Normalized Tone Buffers

Having determined the sample period of a tone, the next step is to extract period-normalized buffers from the sustain region of the tone, one buffer per final sample function. We choose to extract buffers two periods in length from the original tone, evenly distributed across the sustain region. These two-period buffers are then fed through a sample rate conversion filter, yielding a slightly greater than two-period buffer with the desired normalized sample period, as shown in Figure 2.3.

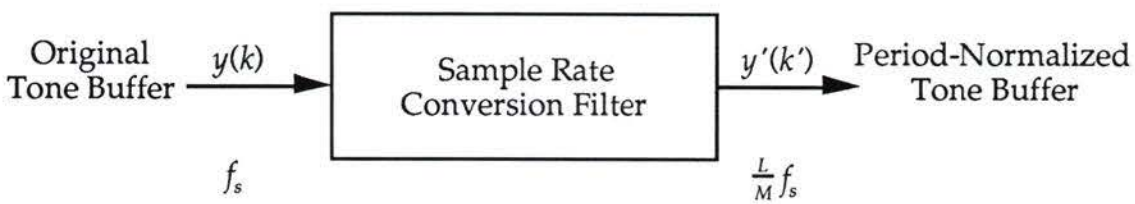

Figure 2.3 - Normalizing Period with Sample Rate Conversion Filter

We have chosen to work with normalized sampled periods equal to either 128 or 256, due to their power-of-two length which is often more convenient in final implementations of the wavetable oscillators (which have lookup tables of the same length). Stapleton and Bass used a normalized sample period of 128.

Period normalization can be viewed as sample rate conversion per the following. If the original tone's sample period K_{tone} is less than the normalized sample period K_p, then to increase the number of sample points per period is equivalent to increasing the sampling frequency f_s at which the tone is sampled. Similarly, if K_{tone} is greater than K_p, then to decrease the number of sample points per period is equivalent to decreasing f_s. Period normalization is equivalent to converting the original sample rate f_s to the new sample rate $\frac{L}{M} f_s$, where the integer $L = K_p$ and the integer $M = K_{tone}$.

A thorough presentation of the theory and implementation of sample rate conversion filters is provided in [32]. Herein, we will focus on implementation. A sample rate conversion filter can be broken into two stages, as shown in Figure 2.4.

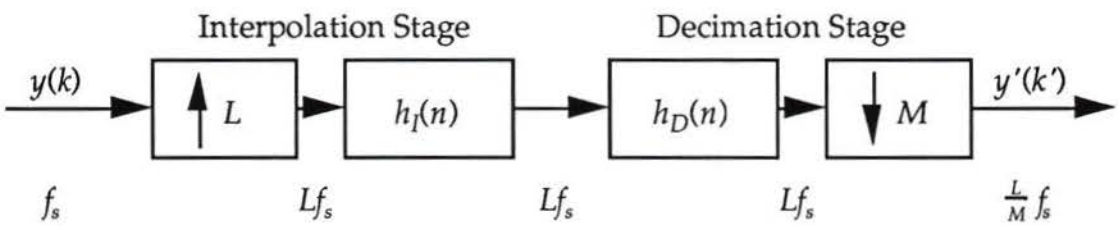

Figure 2.4 - Sample Rate Conversion Filter as Cascaded Interpolator and Decimator

The first stage of the filter performs interpolation, increasing the sampling rate from f_s to Lf_s. The second stage of the filter performs decimation,

decreasing the sampling rate from Lf_s to $\frac{L}{M}f_s$. An interpolating filter operating at the frequency Lf_s with impulse response $h_I(n)$ is at the output of the first stage. A band-limiting decimation filter with impulse response $h_D(n)$, also operating at the frequency Lf_s, is at the input of the second stage. The interpolation stage operates by inserting L-1 zeros between each successive sample of $y(k)$, and then low-pass filtering the zero-interpolated signal. The decimation stage operates by first low-pass filtering the interpolated signal to prevent aliasing after decimation, and then discarding M-1 of every M samples. (The band-limiting decimation filter need not be included if $M<L$, since there is no risk of aliasing for this case.) The interpolation and decimation filter responses can be convolved to yield a single interpolation/decimation filter with impulse response $h(n) = h_I(n) * h_D(n)$.

The cascaded interpolator and decimator was used in the work of Stapleton and Bass. A significantly more efficient filter implementation than shown in Figure 2.4 is realizable. To introduce the more efficient sample rate conversion filter, we will first introduce a more efficient interpolation filter. Figure 2.5 shows a polyphase interpolator with a commutator output. The structure operates as follows. The L polyphase filters $p_0(q)...p_{L-1}(q)$ each operate on the same length Q window of $y(k)$, producing one sample each at a commutator tap. The commutator outputs one sample per tap, incrementing through the taps in steps of one. When the commutator returns from Tap L-1 to Tap 0, the next sample of $y(k)$ is read into the window for the polyphase filters to act on.

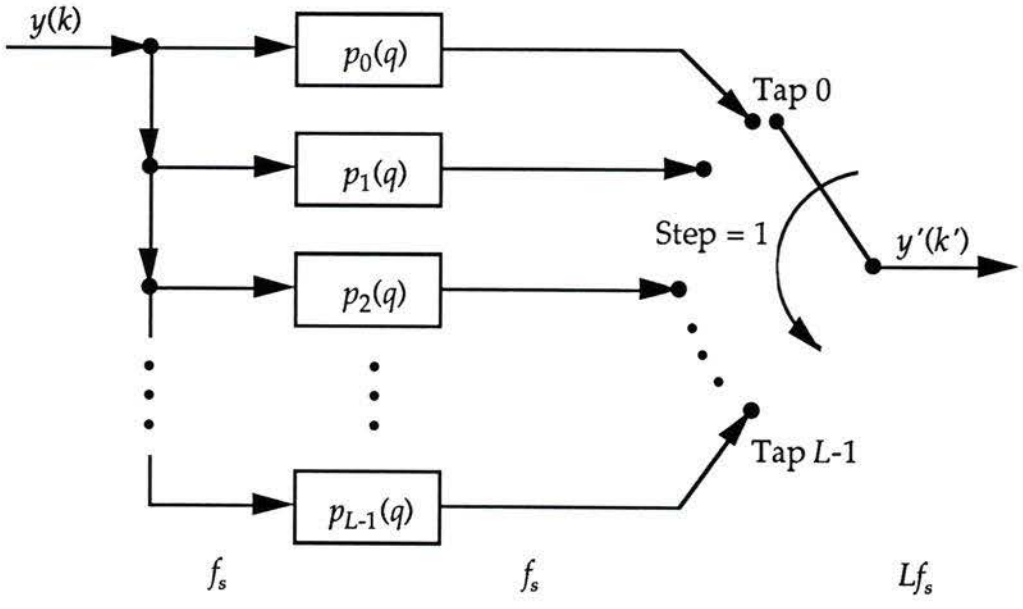

Figure 2.5 - Polyphase Interpolator with Commutator Output

The efficient sample rate conversion filter is just a simple extension of the polyphase interpolator, as shown in Figure 2.6.

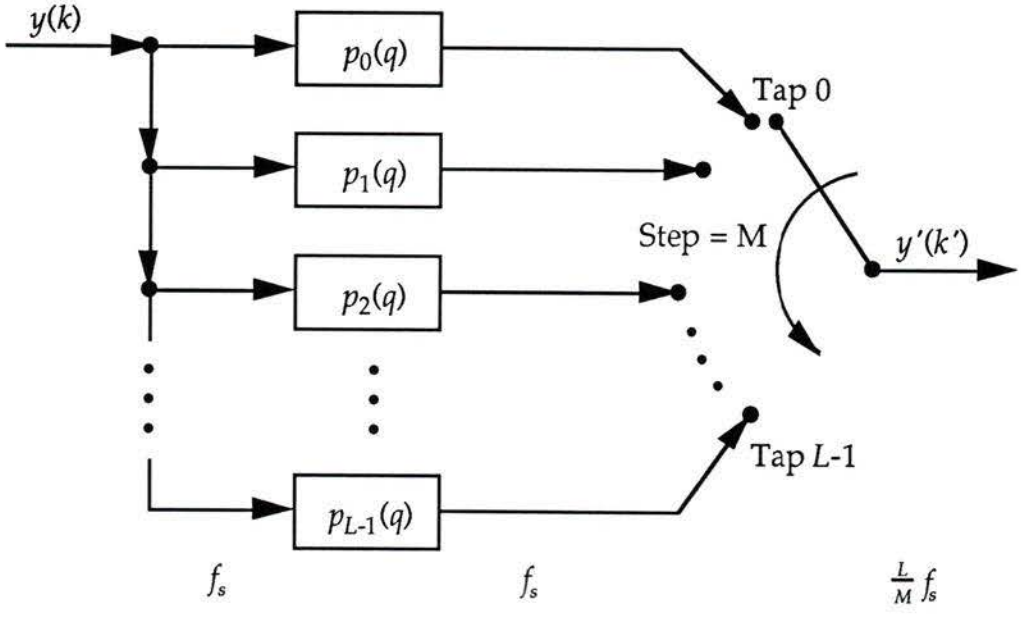

Figure 2.6 - Efficient Sample Rate Conversion Filter

Here the commutator increments through the taps with a step-size of M. Every time the commutator cycles past Tap L-1, the next sample of $y(k)$ is read into the polyphase filters' window. In cases where $M>>L$, then multiple samples of $y(k)$ will be shifted into the window with each commutator tap increment, since a singe increment of M would cycle past Tap L-1 multiple times. For efficiency, only the polyphase filter whose tap will be read should be executed with each commutator tap increment. Note that for M=1, this structure reduces to that of the simple polyphase interpolator.

The remaining step in implementing the sample rate conversion filter of Figure 2.6 is determining the structure and coefficients of the polyphase filters. Typically a linear phase response is desired in the sample rate conversion process. For the sample rate conversion scheme given in Figure 2.4, implementing the interpolation/decimation filter $h(n)$ as a finite impulse response (FIR) structure allows for linear phase. As well, by using an FIR structure for the polyphase filters, the coefficients for these filters can be taken directly from the FIR interpolation/decimation filter $h(n)$.

We have chosen to use the following criteria in designing the filter response $h(n)$. The length of each polyphase filter is set to be a common constant Q. We have typically used a value of Q=6. The polyphase filter length Q can also be thought of as the number of samples from the incoming signal being used to interpolate a value for a single output sample. Each output sample is a weighted sum of Q input samples.

$$y'(k') = \sum_{q=0}^{Q-1} y(k-q)p_i(q) \tag{2-8}$$

The length of the filter response $h(n)$ is set to be $N=LQ$. The filter is designed as a low-pass with a normalized cutoff frequency (in Hz) of

$$f_c = \min\left(\frac{1}{L}, \frac{1}{M}\right), \tag{2-9}$$

where 1 Hz corresponds to half the interpolation sampling frequency Lf_s. Finally, we choose to design the low-pass filter response $h(n)$ with normalized cutoff frequency f_c and length N using a Blackman window. The choice of window is somewhat arbitrary. A Blackman window provides a reasonable tradeoff between maximizing side-lobe attenuation and minimizing main-lobe width for a window's frequency characteristic.

Having designed the interpolation/decimation filter response $h(n)$, the polyphase filter coefficients are extracted as follows.

$$p_i(q) = h(i + Lq), \qquad i = 0,1,\ldots,L-1,\ q = 0,1,\ldots,Q-1 \tag{2-10}$$

Each of the polyphase filters $p_i(q)$ is a decimated version of $h(n)$. The term polyphase arises from the fact that each has a different delay and consequently phase shift. The different delays are compensated by the different times at which each filter's output is sampled, resulting in an overall linear phase response.

2.1.3. Extracting Energy-Normalized Single-Period Sample Functions

The final sample function extraction step involves:

1) cutting a single-period buffer from each period-normalized buffer,
2) passing each single-period buffer cyclically through a low-pass filter,
3) cutting a single-period sample function from the output of the low-pass filter, and

4) normalizing the energy of each single-period sample function.

A more detailed description of the above four steps for extracting one sample function $x(k)$ from a period-normalized buffer $y'(k')$ follows.

A single-period buffer $x'(k)$ of length K_P is cut directly from the middle of $y'(k')$ which is approximately of length $2K_P$. If $x'(k)$ is repeated cyclically, in general there will be a discontinuity from sample $x'(K_P)$ to sample $x'(1)$. To eliminate this discontinuity, $x'(k)$ is passed cyclically through a low-pass filter for three cycles, yielding a smoothed buffer $x_S(k)$.

$$x_S(k+1) = \text{lowpass}\left(x'\left(k \bmod K_P + 1\right)\right), \quad k = 0,1,\ldots,3K_P - 1 \tag{2-11}$$

The cutoff frequency of the low-pass filter should be chosen based on the highest expected playback frequency f_H of notes which will use basis functions derived in part from this sample function. The normalized frequency of a single-period sample function is

$$f_P = \frac{f_s}{K_P}. \tag{2-12}$$

Thus a cutoff frequency of somewhat less than

$$f_c = \frac{f_P}{f_H}\frac{f_s}{2} \tag{2-13}$$

should be used in a linear phase FIR filter of sufficient order to provide sufficient attenuation above f_c for preventing aliasing in synthesis. Note that f_s here refers to the sampling frequency at which tones will be synthesized, which need not be the same as the sampling frequency at which a tone being analyzed was originally sampled.

A single-period sample function $\hat{x}(k)$ (un-normalized in energy) is cut from $x_s(k)$ starting at the first zero-crossing following the first cycle of $x_s(k)$. It is desirable to normalize $\hat{x}(k)$'s energy for two reasons. First, this makes the contribution of each sample function extracted of equal importance in defining the sample function space. Second, when the KLT is applied to a group of sample functions each normalized to unit energy, the resulting eigenvalues sum to one, meaning that each eigenvalue represents the fraction of energy in the sample function space represented by its eigenvector or basis function. This is convenient when analyzing the KLT results. The function $\hat{x}(k)$ is normalized to unit energy as follows. Recalling the interchangeable sample function/sample vector notation used in Chapter 1, Section 1.3.1, refer to $\hat{x}(k)$ and $x(k)$ in their vector forms $\hat{\mathbf{X}}$ and $\mathbf{X}$ respectively. The vector $\hat{\mathbf{X}}$ is normalized to unit energy as

$$\mathbf{X} = \frac{1}{|\hat{\mathbf{X}}|}\hat{\mathbf{X}} \tag{2-14}$$

where $|\hat{\mathbf{X}}|$ is the magnitude of $\hat{\mathbf{X}}$. The sample function form of $\mathbf{X}$, $x(k)$, is the desired sample function, normalized in period, periodically extensible, and normalized to unit energy.

2.2. KL Basis Function Computation for Ensemble of Instrument Tones

The KLT can be applied to sample functions from one or more instrument tones. In the original work of Stapleton and Bass, the choice of what instrument tones to group in an ensemble was performed using an automated classification procedure. The procedure grouped tones which exhibited high inter-correlation. Any one tone in a class or ensemble was well-correlated with every other tone in the class.

We choose to not use a classification procedure, instead limiting ensembles to tones from one or at most two instruments. If it is desired to group tones from more than two instruments, it is usually reasonable to assume that there will be good correlation in a family of instruments, such as single-reed woodwinds, double-reed woodwinds, brass and strings.

Extracting the KL basis functions for an ensemble of sample functions involves three steps. First, the tones from which to form the ensemble of sample functions must be chosen. This we allow a human user to do arbitrarily. Second, the sample functions are aligned in phase. Third, the KL basis functions are computed for the phase-aligned ensemble of sample functions. The second and third steps are described in more detail below.

2.2.1. Phase Aligning Sample Functions

Figure 2.7 shows twelve sample functions taken from three different trumpet tones, four sample functions per tone.

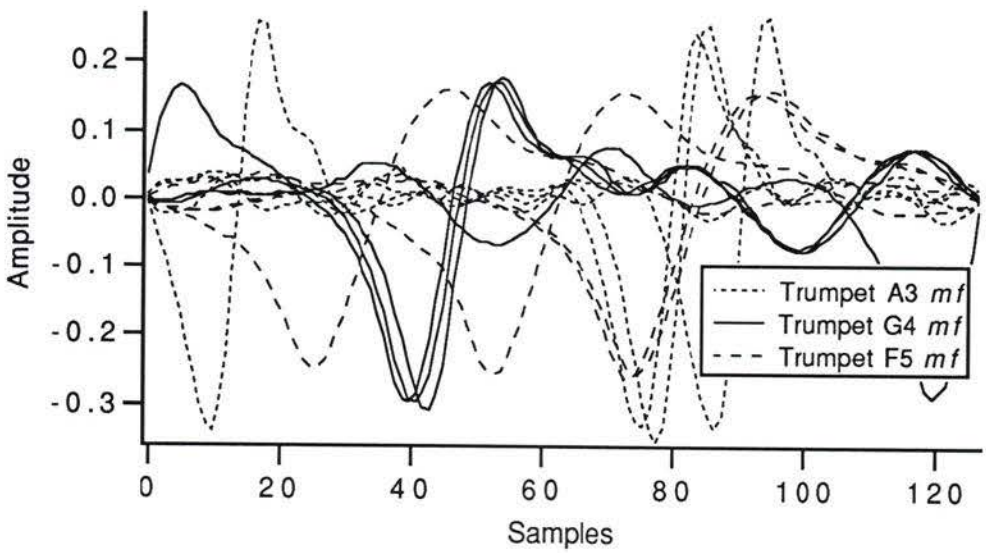

Figure 2.7 - Ensemble of 12 Sample Functions from 3 Trumpet Tones Before Phase Alignment

Figure 2.8 shows the same ensemble of twelve sample functions after they have been aligned in phase.

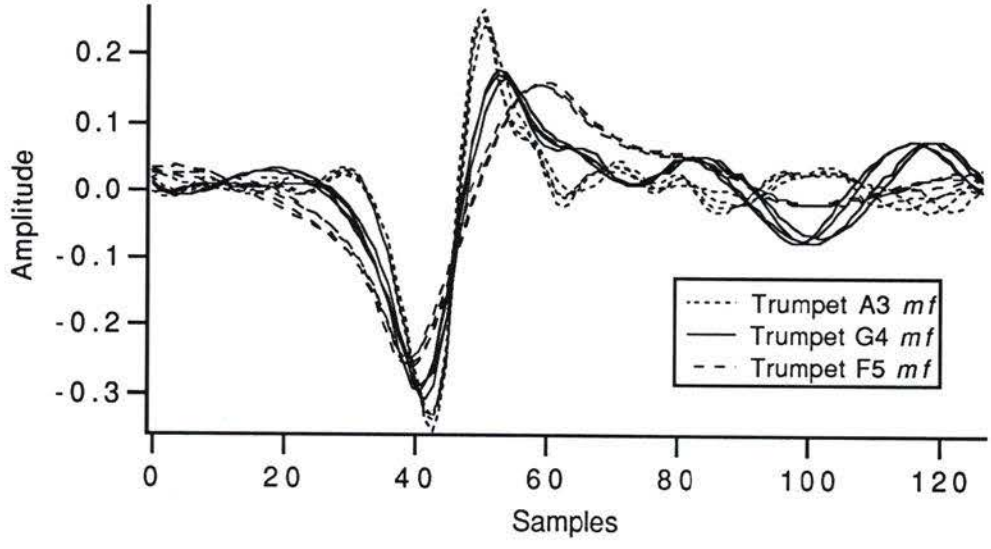

Figure 2.8 - Aligned Ensemble of 12 Sample Functions from 3 Trumpet Tones

It is important to align the sample functions in phase in order to maximize the inter-correlation for the ensemble. The higher the correlation between sample functions, the more energy that will be concentrated in the first few KL basis functions. Table 2.2 provides a comparison of the KL basis function weights computed first for the unaligned sample functions, then the aligned sample functions. For the unaligned sample functions, the first eight basis functions have weights greater than .01. Each of the first eight basis functions contain more than 1% of the total energy in the sample function space. As well, the highest energy basis function only contains about 30% of the sample function space energy. For the aligned sample functions, only the first three basis functions have weights greater than .01. The highest energy

basis function contains approximately 85% of the sample function space energy.

Weights for Unaligned Sample Functions	Weights for Aligned Sample Functions
0.3010	0.8512
0.2528	0.1137
0.1402	0.0273
0.1098	0.0031
0.0822	0.0019
0.0461	0.0013
0.0352	0.0006
0.0152	0.0003
0.0097	0.0002
0.0076	0.0001
0.0003	0.0001
0.0002	0.0000

Table 2.2 - Comparison of Basis Function Weights for Unaligned and Aligned Ensemble of 12 Sample Functions from 3 Trumpet Tones

The procedure for aligning the sample functions in phase is as follows. Consider a set of M unaligned sample functions $x_i(k)$ with $i = 1,2,\ldots,M$ and $k = 1,2,\ldots,K_p$. The best sample function alignment occurs when the energy in the dimension of the first eigenvector $\boldsymbol{\beta}_1$ of the sample functions' correlation matrix $\mathbf{R}_x$ is maximized. Maximizing the eigenvalue λ_1 of the first eigenvector of $\mathbf{R}_x$ is equivalent to maximizing its energy. The steps in maximizing λ_1 follow.

1) Compute λ_1 and $\boldsymbol{\beta}_1$ for the current alignment of the sample functions. (Initially the sample functions are completely unaligned.)
2) Compare the value of λ_1 just computed to the previous value of λ_1. (On the first pass through these steps, the previous value of λ_1 is initialized to 0.) If the value of λ_1 has increased by more than some threshold, continue with the phase alignment procedure; otherwise halt the procedure. We have typically used an increase threshold of .0001 with the maximum possible value of λ_1 being 1.
3) Phase align each of the sample functions with $\boldsymbol{\beta}_1$. For each $x_i(k)$ the alignment with $\boldsymbol{\beta}_1$ is achieved by choosing a phase shift k_a in the range $0,1,\ldots,K_P-1$ which maximizes the expression:

$$\sum_{k=1}^{K_P} x_i\big((k+k_a-1)\bmod K_P \;+\; 1\big)\beta_1(k). \tag{2-15}$$

Each sample function $x_i(k)$ is aligned using its phase shift k_a.

$$x_i(k) = x_i\big((k+k_a-1)\bmod K_P \;+\; 1\big), \quad k=1,2,\ldots,K_P \tag{2-16}$$

The above three steps are repeated until either the condition in step 2 for halting has been satisfied or some maximum number of iterations have occurred. Typically, phase alignment occurs in less than 10 iterations. The phase alignment illustrated in Figures 2.7 and 2.8 requires 3 iterations.

The first step in the procedure requires the computation of the first eigenvalue and eigenvector of the sample functions' correlation matrix. Stapleton and Bass present an efficient method for this computation. For convenience, we have chosen to use the Singular Value Decomposition technique described in the next section for computing all the eigenvalues and eigenvectors of the correlation matrix.

2.2.2. Applying the KLT to the Sample Functions

One approach to computing the KLT is outlined in Chapter 1, Section 1.3.1. Based on the definition of the KLT, the correlation matrix $\mathbf{R}_x$ for a set of M sample functions is estimated as

$$\mathbf{R}_x \approx \frac{1}{M}\sum_{i=1}^{M}\mathbf{x}_i\mathbf{x}_i^T. \tag{2-17}$$

(Recall that the K-by-1 vector $\mathbf{x}_i$ is the sample vector form of the length K_P sample function $x_i(k)$.) Then the eigenspace of $\mathbf{R}_x$ is computed to satisfy (2-18) and (2-19).

$$\mathbf{R}_x\boldsymbol{\beta}_n = \lambda_n\boldsymbol{\beta}_n, \qquad n = 1,2,\ldots,K_P \tag{2-18}$$

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0. \tag{2-19}$$

Methods for computing the eigenvalues and eigenvectors of $\mathbf{R}_x$ are presented in [33]. A software package such as *Matlab* [34] includes an efficient function for computing eigenvalues and eigenvectors.

The KLT can also be computed using the Singular Value Decomposition (SVD). The SVD is preferred because it is more computationally efficient and numerically stable than eigenanalysis. The SVD is used to compute the KLT as follows. First, define a data matrix of sample functions $\mathbf{X}$ with dimension K_P-by-M.

$$\mathbf{X} = \left[\mathbf{x}_1 \mid \mathbf{x}_2 \mid \cdots \mid \mathbf{x}_M\right] \tag{2-20}$$

where $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M$ are M ensemble sample functions $\mathbf{x}_i$. Equation (2-17) can be re-written in matrix form as

$$\mathbf{R}_x \approx \frac{1}{M}\mathbf{X}\mathbf{X}^T. \tag{2-21}$$

Instead of applying an eigenanalysis routine to the correlation matrix $\mathbf{R}_\mathbf{x}$, the SVD is applied directly to the data matrix $\mathbf{X}$.

The SVD expands $\mathbf{X}$ as follows.

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T,$$

$$\mathbf{U} = \left[\mathbf{u}_1 \mid \mathbf{u}_2 \mid \cdots \mid \mathbf{u}_{K_P} \right], \quad (K_P\text{-by-}K_P),$$

$$\mathbf{S} = \left[\begin{array}{ccc|c} \sigma_1 & & \mathbf{0} & \\ & \sigma_2 & & \mathbf{0} \\ \mathbf{0} & & \ddots & \\ & & & \sigma_R \\ \hline & \mathbf{0} & & \mathbf{0} \end{array}\right], \quad (K_P\text{-by-}M), \tag{2-22}$$

$$\mathbf{V} = \left[\mathbf{v}_1 \mid \mathbf{v}_2 \mid \cdots \mid \mathbf{v}_M \right], \quad (M\text{-by-}M).$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ are orthogonal. The K_P-by-1 orthogonal vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{K_P}$ are called the left singular vectors of $\mathbf{X}$. The M-by-1 orthogonal vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_M$ are called the right singular vectors of $\mathbf{X}$. The matrix $\mathbf{S}$ is all zeros except for the non-zero singular values $\sigma_1, \sigma_2, \ldots, \sigma_R$. R is simply the rank of the original data matrix $\mathbf{X}$ and is restricted as follows.

$$R \le \min(K_P, M) \tag{2-23}$$

The vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{K_P}$ are also the eigenvectors of $\mathbf{X}\mathbf{X}^T$, while the squared singular values $\sigma_1^2, \sigma_2^2, \ldots, \sigma_R^2$ are the non-zero eigenvalues of $\mathbf{X}\mathbf{X}^T$. These last two properties are used to compute the KLT. Using these properties, the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_{K_P}$ and eigenvectors $\boldsymbol{\beta}_1, \boldsymbol{\beta}_2, \ldots, \boldsymbol{\beta}_{K_P}$ of $\mathbf{R}_\mathbf{x}$ are computed as follows.

$$\begin{aligned} \lambda_n &= \tfrac{1}{M}\sigma_n^2, & n &= 1, 2, \ldots, R \\ \lambda_n &= 0, & n &= R+1, R+2, \ldots, K_P \\ \boldsymbol{\beta}_n &= \mathbf{u}_n, & n &= 1, 2, \ldots, K_P \end{aligned} \tag{2-24}$$

The equations of (2-24) are a result of the fact that if $\lambda_1,\lambda_2,\ldots,\lambda_{K_p}$ and $\boldsymbol{\beta}_1,\boldsymbol{\beta}_2,\ldots,\boldsymbol{\beta}_{K_p}$ are respectively the eigenvalues and eigenvectors of a K_p-by-K_p matrix $\mathbf{A}$, then $c\lambda_1,c\lambda_2,\ldots,c\lambda_{K_p}$ and $\boldsymbol{\beta}_1,\boldsymbol{\beta}_2,\ldots,\boldsymbol{\beta}_{K_p}$ are respectively the eigenvalues and eigenvectors of $c\mathbf{A}$, where c is a scalar constant multiplying the matrix $\mathbf{A}$. As defined in (2-21), $\mathbf{R}_x$ is simply $\mathbf{X}\mathbf{X}^T$ multiplied by a scalar constant $\frac{1}{M}$.

Methods for computing the SVD are presented in [33]. The *Matlab* software package includes an efficient function for computing the SVD. The routine can either solve for all of **U**, **S** and **V**, or just perform an "economy size" SVD, computing only **U** and **S**. The latter is favored for computing the KLT, as there is no need to compute the right singular vectors in **V**. As well, the "economy size" SVD only computes the first M columns of **U** if $M<K_p$. This is helpful, since typically we use less sample functions in an ensemble than the sample function length K_p.

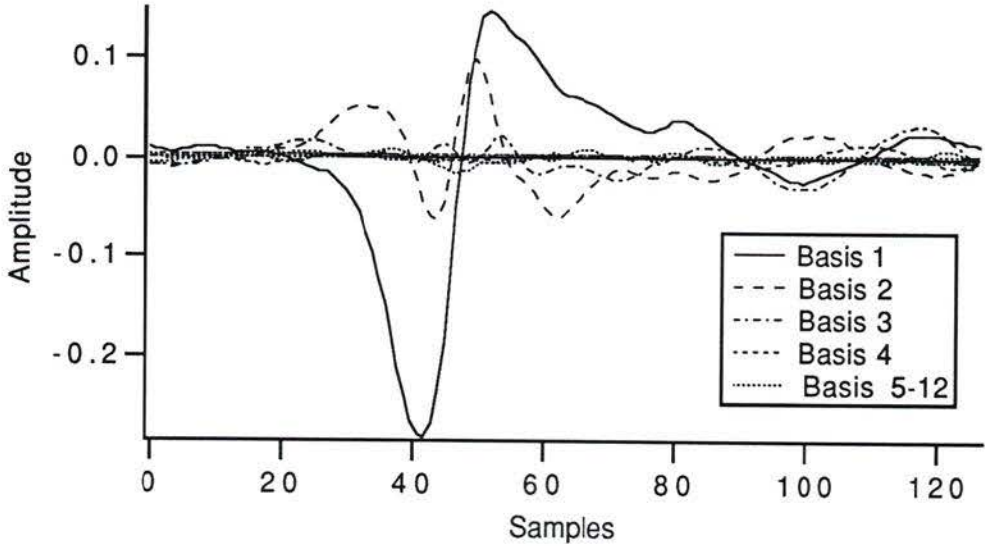

Figure 2.9 - Weighted Basis Functions Extracted from 3 Trumpet Tones

Figure 2.9 shows the basis functions computed for the phase-aligned sample functions of Figure 2.8. Each of the basis functions have been scaled by the square root of their associated eigenvalues or weights, to reflect their relative share of the energy in the sample function space. The weights are listed in the second column of Table 2.2 above. Only the first three or four basis functions contain significant energy. The first four basis functions are presented separately and unweighted in Figure 2.10.

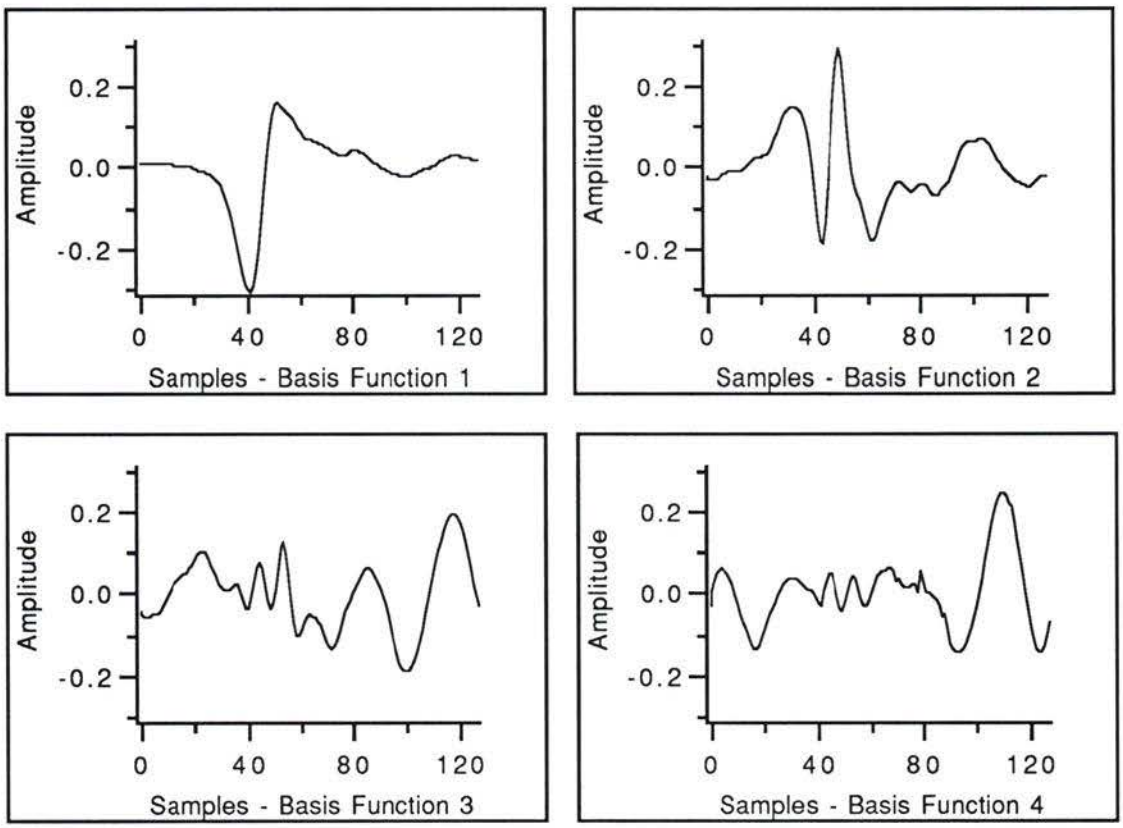

Figure 2.10 - First 4 Basis Functions Extracted from 3 Trumpet Tones

2.3. Extraction of Synthesis Parameters

Having determined the basis function set for an ensemble of sample functions from one or more instrument tones, the final step of extracting

synthesis parameters is performed. Recall from Chapter 1, Section 4 that the synthesis equation using KL basis functions is

$$s(k) = \sum_{n=1}^{N} A_n(k)\beta_n(\phi(k)). \tag{2-25}$$

In order to perform synthesis of a single tone, a set of N time-varying amplitude envelopes or functions $A_n(k)$ and a single phase function $\phi(k)$ must be determined. The amplitude envelopes and phase function will be different for each tone represented in the sample function ensemble. The first step in extracting synthesis parameters is simply to determine $A_n(k)$ and $\phi(k)$ for each tone so as to re-synthesize the original tone. A second optional step is to compress the information required to represent the amplitude envelopes and phase function for a tone.

2.3.1. Determining the Amplitude Envelopes and Phase/Frequency Functions for a Tone

The most inefficient representation of the amplitude envelopes and phase function for a tone involves determining and storing values of $A_n(k)$ and $\phi(k)$ for each successive value of the discrete-time index k. Each amplitude envelope plus the phase function would have as many discrete values as the original sampled tone. Clearly, this is less efficient than just storing the original sampled tone. Instead, we choose to determine values of the amplitude envelopes and phase function at a small number of points spread out over the tone's duration. For a high resolution approximation, we compute values for the amplitude envelopes and phase function once every two sample periods throughout the sustain and decay regions. In the attack region, values are determined every half period. All values for the amplitude

envelopes and phase function for sample points between the computed values are interpolated.

The following procedure is used to compute the values for the amplitude envelopes and phase function over the full duration of a tone.

1) The period-normalized basis functions are converted in length to the sample period of the tone being analyzed.
2) The values of the phase function and the amplitude envelopes are determined at the beginning of the sustain region.
3) The values for the phase function and the amplitude envelopes are calculated from the beginning of the sustain region to the end of the tone.
4) The values for the phase function and the amplitude envelopes are calculated from the beginning of the sustain region to the beginning of the tone. It is best to work backward from the beginning of the sustain region instead of forward from the beginning of the tone because the phase function estimate at the beginning of the sustain region is much more reliable than one computed at the onset of the tone.

Step 1 of the procedure is performed to allow for efficient vectorized computation of the phase function and amplitude envelopes in Steps 2-4. This is a departure from the work of Stapleton and Bass, who did not vectorize the computation. Each unit-energy basis function represented by K_p discrete points is converted to a unit-energy basis function represented by K_{tone} discrete points. Instead of using a sample rate conversion filter for this process, we choose to use a cubic spline data interpolation function in order to better preserve the phase in the re-sampled basis functions. The cubic spline interpolation does not allow for specification of the frequency effects of the

conversion, but this is not deemed important since the interpolated basis functions will only be used in this analysis procedure, and not for tone synthesis. The cubic spline interpolation algorithm is described in many texts, including [35] by Burden and Faires. The *Matlab* software package includes a spline interpolation function.

The result of performing spline interpolation is as follows. Each original basis function vector $\boldsymbol{\beta}_n$ of length K_p is converted to an interpolated basis function vector $\tilde{\boldsymbol{\beta}}_n$ of length K_{tone}. The K_{tone} elements of $\tilde{\boldsymbol{\beta}}_n$ correspond to the K_p elements of $\boldsymbol{\beta}_n$ as

$$\tilde{\boldsymbol{\beta}}_{n,k} = \boldsymbol{\beta}_{n,k'} \ , \quad k = 1,2,\ldots,K_{tone}. \tag{2-26}$$

The index k' in (2-26) is defined as

$$k' = \begin{cases} 1, & k = 1 \\ \dfrac{K_p}{K_{tone}}(k-1) + 1, & k = 2,3,\ldots,K_{tone} \end{cases}. \tag{2-27}$$

Only the first value of $k'$is guaranteed to be integer. All non-integer values of k' indicate that $\tilde{\boldsymbol{\beta}}_{n,k}$ should be interpolated using a cubic polynomial from the four nearest integer-indexed elements to k' in $\boldsymbol{\beta}_n$. The condition that $\tilde{\boldsymbol{\beta}}_{n,1} = \boldsymbol{\beta}_{n,1}$ guarantees that the interpolated basis function is aligned in phase with the original basis function.

The interpolated basis function $\tilde{\boldsymbol{\beta}}_n$ is no longer of unit energy. To force $\tilde{\boldsymbol{\beta}}_n$ to be unit energy, we simply perform the normalization

$$\tilde{\boldsymbol{\beta}}_n = \frac{1}{|\tilde{\boldsymbol{\beta}}_n|}\tilde{\boldsymbol{\beta}}_n. \tag{2-28}$$

Steps 2-4 involve similar procedures. First in Step 2, the phase function value is determined for the beginning of the sustain region. We determine the phase function value by comparing a window of the tone with the first basis function. The first basis function is used because it represents the most tone energy in the basis function set. Define the point in the tone marking the beginning of the sustain region as k_0. Extract a window of K_{tone} samples from the tone beginning at k_0.

$$x_{win}(k) = tone(k_0 + k - 1), \qquad k = 1,2,\ldots,K_{tone} \tag{2-29}$$

The phase value at k_0 is the offset k_a in the range $k_a = 1,2,\ldots,K_{tone}$ which maximizes the expression:

$$\sum_{k=1}^{K_{tone}} \tilde{\beta}_1\big((k + k_a - 2) \bmod K_{tone} + 1\big) x_{win}(k). \tag{2-30}$$

We then set the phase function at k_0 as

$$\phi(k_0) = k_a. \tag{2-31}$$

This technique provides only K_{tone} discrete possibilities for the phase. Stapleton and Bass performed a quadratic interpolation of three discrete phases to arrive at a non-integer value for the phase; however, this is more computationally expensive, and does not lend itself to vectorized computation of the amplitude envelope values based on an integer phase index. We think it is sufficient to only determine integer values for the phase at any point in the tone, as it is not so much the exact phase which we are concerned with, but the change in phase over the course of the tone.

The amplitude envelope values for the beginning of the sustain region are computed using the maximizing offset k_a.

$$A_n(k_0) = \sum_{k=1}^{K_{tone}} \tilde{\beta}_n\big((k + k_a - 2) \bmod K_{tone} + 1\big) x_{win}(k), \quad n = 1, 2, \ldots, N \tag{2-32}$$

In Step 3, the basic procedure of Step 2 is iterated at points $k_1, k_2, \ldots, k_{end}$ across the sustain and decay regions of the tone. The locations of the points are defined as

$$k_i = k_0 + 2iK_{tone}. \tag{2-33}$$

The point k_{end} is the last sample in the tone. The procedure of Step 2 is varied in one aspect. The range of values k_a evaluated to determine $\phi(k_i)$ at each point is restricted based on the assumption that the phase does not change significantly over just two sample periods of the tone. In determining $\phi(k_i)$ Equation (2-30) is evaluated for a maximum over the range

$$k_a = \phi(k_{i-1}) - \frac{K_{tone}}{16}, \ldots, \phi(k_{i-1}) + \frac{K_{tone}}{16}. \tag{2-34}$$

Note that the arithmetic in (2-34) is performed modulo K_{tone} to yield values of k_a in the range $1, 2, \ldots, K_{tone}$. The range of values k_a may wrap around from K_{tone} to 1 at times.

Step 4 is similar to Step 3, except that points in the attack region $k_{-1}, k_{-2}, \ldots, k_{begin}$ are computed backwards from the beginning of the sustain region to the beginning of the tone. The locations of the points are defined as

$$k_{-i} = k_0 - 0.5iK_{tone}. \tag{2-35}$$

Here equation (2-30) is evaluated for a maximum over the range

$$k_a = \left(\phi(k_{i-1}) - \frac{K_{tone}}{2}\right) - \frac{K_{tone}}{16}, \ldots, \left(\phi(k_{i-1}) - \frac{K_{tone}}{2}\right) + \frac{K_{tone}}{16}. \tag{2-36}$$

A number of arbitrary factors are involved in Steps 3 and 4. These are:

1) period step rate of 2 through the sustain and decay region,
2) period step rate of 0.5 through the attack region, and
3) phase search window size of $\pm K_{tone}/16$.

Stapleton and Bass used an adaptive procedure which varied the step rates and search window size based on how fast the phase was changing at any point in the tone. The values we chose are conservative enough to not lose track of the phase, though at times we may be computing the phase more often than necessary when it is not rapidly changing. Using a non-adaptive algorithm is preferred with a package such as *Matlab* where mathematical computations are performed most efficiently and conditional expressions required for adaptation are performed less efficiently. The non-adaptive algorithm is also simpler to implement.

After completing the above procedure, the amplitude envelopes and phase function are defined at the points $k_{begin},\ldots,k_{-1},k_0,k_1,\ldots,k_{end}$. The amplitude envelopes' values need to be scaled to correspond to the original period-normalized basis functions.

$$A_n(k_i)=\frac{A_n(k_i)}{|\tilde{\beta}_n|} \tag{2-37}$$

It is also useful to convert the phase function to an instantaneous frequency function. This is a two step process. The phase function is first unwrapped, then converted to a frequency function. The phase function computed in the procedure above is a wrapped phase function, in that it only takes on values between 1 and K_{tone}. An unwrapped phase function $\phi'(k_i)$ is derived from the wrapped phase function $\phi(k_i)$ as follows. First, let $\phi'(k_0)=\phi(k_0)$. The values of $\phi'(k_i)$ for $i=1,2,\ldots,end$ are computed as

$$\phi'(k_i) = \phi'(k_{i-1}) + 2K_{tone} + \Delta\big(\phi(k_i), \phi(k_{i-1})\big). \tag{2-38}$$

Note that the function $\Delta\big(\phi(k_i), \phi(k_{i-1})\big)$ is computed based on the size of the phase search window $K_{tone}/16$. If $\left|\phi(k_i) - \phi(k_{i-1})\right| \le K_{tone}/16$ then

$$\Delta\big(\phi(k_i), \phi(k_{i-1})\big) = \phi(k_i) - \phi(k_{i-1}). \tag{2-39}$$

If $\left|\phi(k_i) - \phi(k_{i-1})\right| > K_{tone}/16$ then

$$\Delta\big(\phi(k_i), \phi(k_{i-1})\big) = \begin{cases} \phi(k_i) - \phi(k_{i-1}) + K_{tone}, & \phi(k_i) < \phi(k_{i-1}) \\ \phi(k_i) - \phi(k_{i-1}) - K_{tone}, & \phi(k_i) > \phi(k_{i-1}) \end{cases}. \tag{2-40}$$

The values of $\phi'(k_i)$ for $i = -1, -2, \ldots, begin$ are computed as

$$\phi'(k_i) = \phi'(k_{i+1}) - \Delta\big(\phi(k_i), \phi(k_{i+1})\big), \tag{2-41}$$

$$\Delta\big(\phi(k_i), \phi(k_{i+1})\big) = \begin{cases} \phi(k_{i+1}) - \phi(k_i), & \phi(k_i) < \phi(k_{i+1}) \\ \phi(k_{i+1}) - \phi(k_i) + K_{tone}, & \phi(k_i) > \phi(k_{i+1}) \end{cases}. \tag{2-42}$$

The frequency function is derived from the unwrapped phase function as

$$f(k_i) = \frac{\phi'(k_{i+1}) - \phi'(k_i)}{k_{i+1} - k_i} \cdot \frac{f_s}{K_{tone}}. \tag{2-43}$$

Figure 2.11 shows the first four basis function amplitude envelopes, wrapped and unwrapped phase functions, and frequency function for the Trumpet G4 tone represented in the ensemble example of the previous section. There are a number of features in the graphs which should be considered.

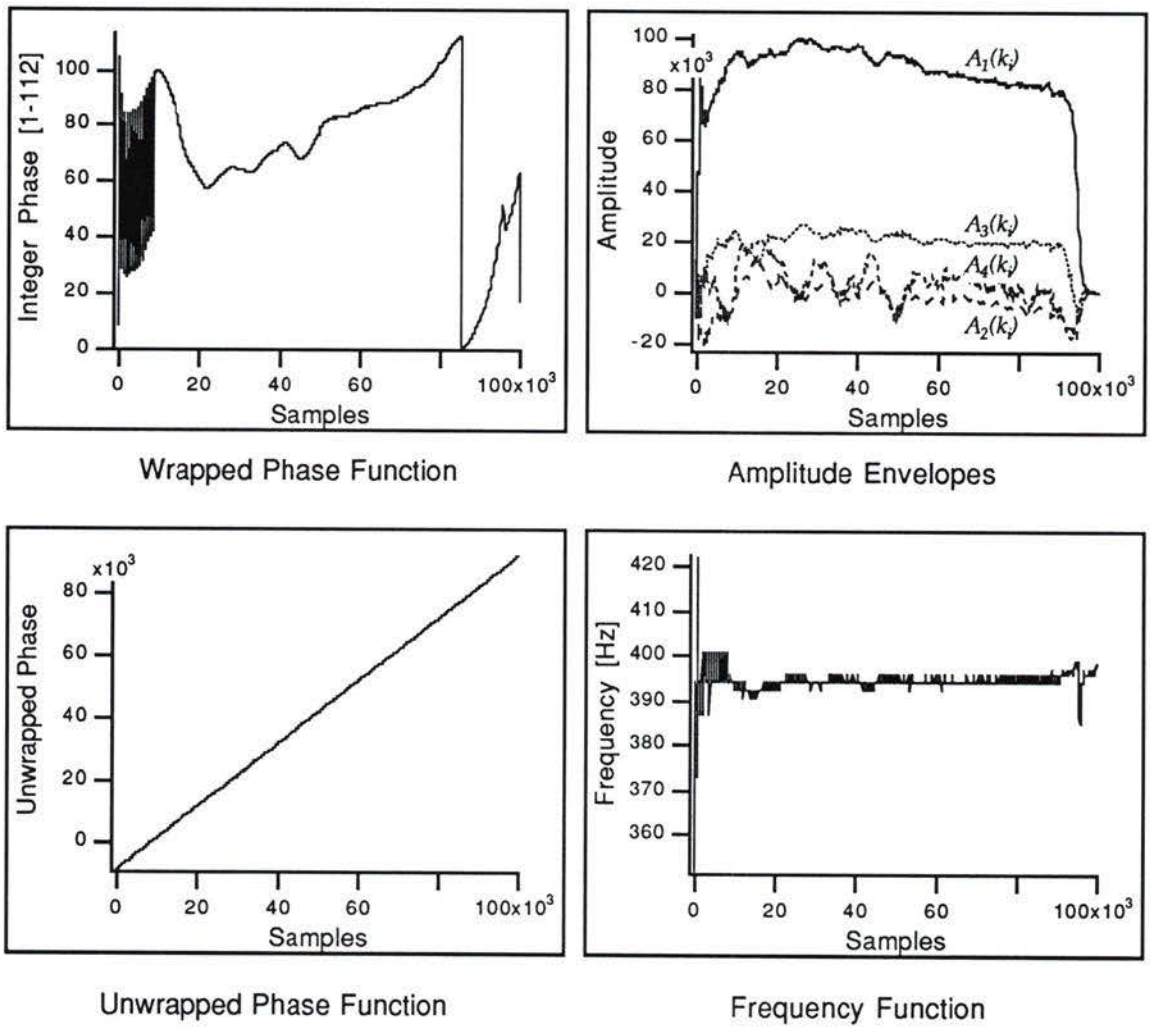

Figure 2.11 - Amplitude Envelopes, Phase and Frequency Functions for a Trumpet Tone

First, during the tone's attack, the unwrapped phase oscillates between values a half-period apart. This is due to determining the phase every half period during the attack. Second, the frequency envelope tends to jitter between a number of discrete points. It is not smooth. This is due to the restriction of only using integer phase values. Clearly, the actual frequency

lies between two discrete frequencies. It would be useful to pass the frequency envelope through a smoothing function to eliminate the jitter. (An effective smoothing function re-computes each frequency envelope sample as a weighted average of itself and a small number of envelope values before and after. We have used a Hamming window centered at the point to be re-computed to define the weights for averaging the current envelope value with earlier and later values. The Hamming window places the most importance on the current envelope value and increasingly less importance on envelope values further away. Note that the sum of all the Hamming window defined weights should be 1.) Third, notice that we allow the amplitude envelopes to take on negative values, which corresponds to an inversion of the basis functions. Finally, notice that the amplitude envelope for basis function 3 is larger than the amplitude envelope for basis function 2. For this particular tone in the ensemble, the third basis function contains more energy corresponding to the tone, than the second. On the average, amplitude envelope levels should decrease with increasing basis function number. But there are occasional exceptions.

Re-synthesis of a tone based on the amplitude envelopes and frequency function computed above is as follows. First define the derivative functions of $A_n(k_i)$ and $f(k_i)$.

$$\Delta A_n(k_i) = \frac{A_n(k_{i+1}) - A_n(k_i)}{k_{i+1} - k_i} \tag{2-44}$$

$$\Delta f(k_i) = \frac{f(k_{i+1}) - f(k_i)}{k_{i+1} - k_i} \tag{2-45}$$

The derivative functions $\Delta A_n(k_i)$ and $\Delta f(k_i)$ are used to update the amplitude envelopes and phase function from sample to sample. For the first tone sample, we initialize $A_n(0) = A_n(k_{begin})$, $\phi(0) = \phi(k_{begin})$ and $f(0) = f(k_{begin})$. Then each successive amplitude envelope and phase function value is computed using the following three equations.

$$A_n(k) = A_n(k-1) + \Delta A_n(k_i), \quad k_i \le k < k_{i+1} \tag{2-46}$$

$$f(k) = f(k-1) + \Delta f(k_i), \quad k_i \le k < k_{i+1} \tag{2-47}$$

$$\phi(k) = \phi(k-1) + \left(f(k) - \frac{\Delta f(k_i)}{2} \right) \frac{K_p}{f_s}, \quad k_i \le k < k_{i+1} \tag{2-48}$$

The data needed to re-synthesize a tone has been reduced to N basis functions, the initial values $A_n(0)$, $\phi(0)$ and $f(0)$, the derivative functions $\Delta A_n(k_i)$ and $\Delta f(k_i)$, and the breakpoints $k_{begin}, \ldots, k_{-1}, k_0, k_1, \ldots, k_{end}$.

2.3.2. Approximating the Amplitude Envelopes and Frequency Function

It is desirable to reduce the number of points specifying the amplitude envelopes and frequency function determined using the procedure outlined in the previous section. Figure 2.12 illustrates how just a few points connected by linear segments reasonably approximate an amplitude envelope. This technique has been applied to the amplitude envelopes and frequency functions of Fourier-based additive synthesis with good results [36]. A small number of well-chosen piecewise-linear segments used in place of the high resolution amplitude envelopes and frequency functions still yields realistic re-synthesis of tones.

A number of techniques exist for approximating the amplitude envelopes and frequency function. One simple approach is to let a human user simply "eyeball" an approximation for the envelopes. The points to retain from the envelopes can be chosen with some graphical input technique. This was the technique used to generate the approximation shown in Figure 2.12. The problem with this technique is that it is not automated.

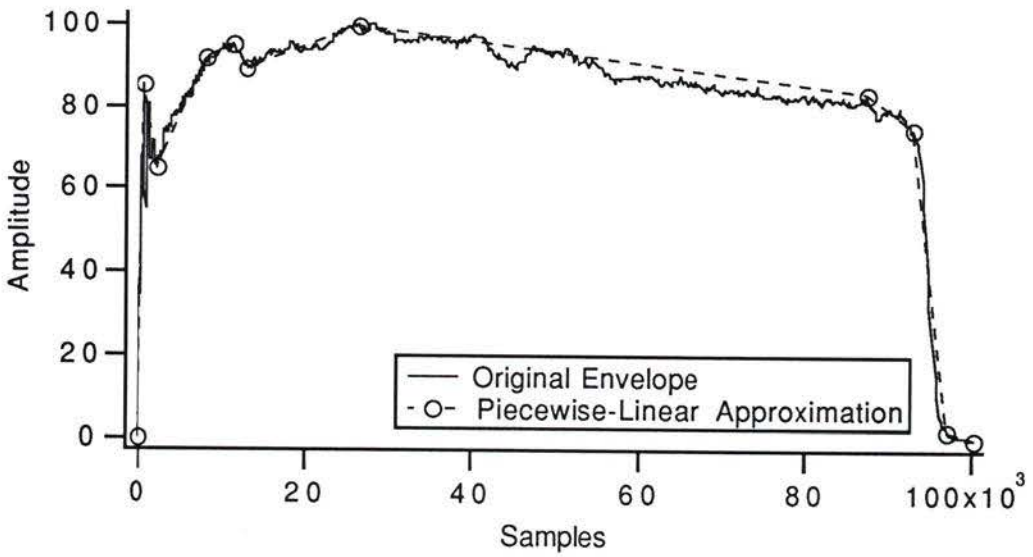

Figure 2.12 - Approximation to Amplitude Envelope of the First Basis Function for a Trumpet Tone

An article by Strawn presents a number of techniques for automatic approximation of the amplitude envelopes and frequency functions used in Fourier-based additive synthesis [37]. These techniques are directly applicable to KL-based synthesis. We will not describe the techniques in detail here, but simply provide an overview. One technique is called the Adjust Algorithm. The technique is relatively simple to implement, and we have used it. The technique involves choosing a set number of points to approximate the envelope with. A first guess is made at an approximation and then iterations

are performed to balance the approximation error between adjacent linear segments. The problem with this technique is that it only performs local error-minimization between adjacent segments, and does not try to globally reduce the approximation error over all segments.

Strawn presents a technique which does globally reduce the approximation error. It is based on syntactic pattern recognition principles. This technique is more difficult to implement, particularly in a numerical-based software package like *Matlab*. We have not used it. Further work on KL-based synthesis should include using a better technique such as Strawn's syntactic method to approximate the amplitude envelopes and frequency functions.

Synthesis using approximated amplitude envelopes and frequency functions is implemented in a similar manner to that described in the preceding section. The only difference is each amplitude envelope and frequency function is uniquely approximated, and therefore each have their own set of breakpoints.

2.4. Implementation of the Analysis/Synthesis Procedure

A menu-driven implementation of the analysis/synthesis procedure described in this chapter has been implemented in the *Matlab* software package. A main menu, illustrated in Figure 2.13, allows the user to move about between the various procedure steps in an arbitrary manner. The source code for the analysis/synthesis software is included in Appendix A. The order of the code follows that of the main menu.

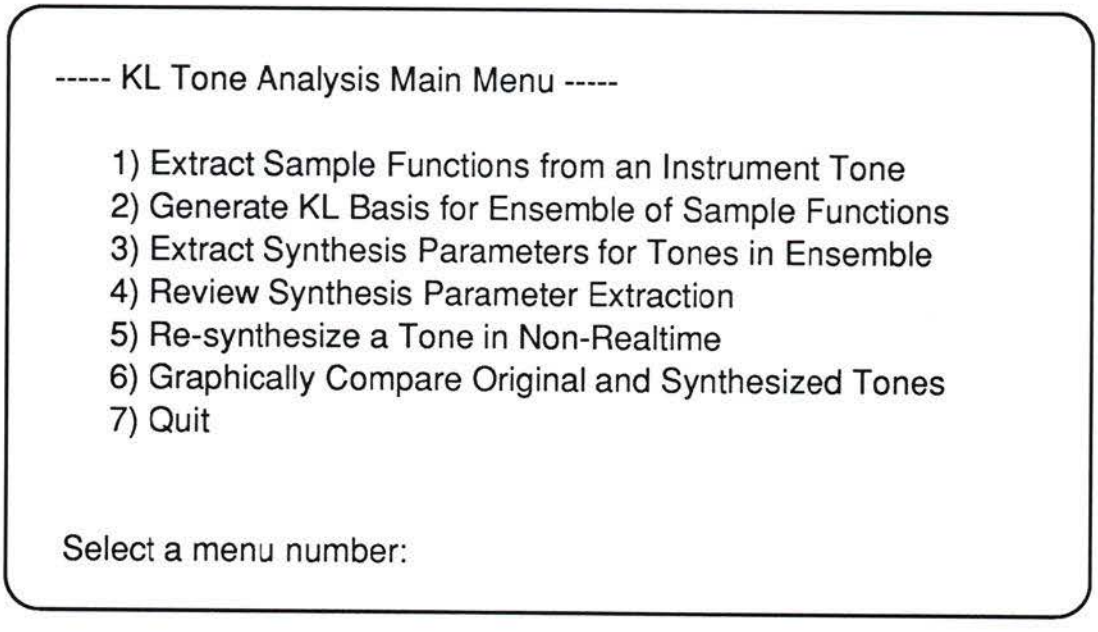

Figure 2.13 - Main Menu of KL Musical Tone Analysis/Synthesis Software

CHAPTER 3
ANALYSIS/SYNTHESIS RESULTS FOR PARTICULAR INSTRUMENTS

In this chapter we first present an overview of the results which Stapleton and Bass achieved analyzing and synthesizing one tone from each of fourteen instruments grouped into highly correlated classes. Then we present our results for analyzing and synthesizing tones from one instrument at a time.

3.1. Previous Analysis/Synthesis Results

Stapleton and Bass chose tones from string, woodwind, brass and mallet instruments. The instrument and pitch for each tone are given in Table 3.1.

Instrument	Pitch [Hz]	Note
Diapason	131	C3
Flute	487	B4
Guitar	188	G3
Marimba	256	C4
Violin	342	F4
Plucked Violin	288	D4
Trumpet	449	A4
French Horn	261	C4
Trombone	211	G#3
Oboe	261	C4
Bassoon	87	F2
Alto Saxophone	305	D#4
Tenor Saxophone	232	A#3
Clarinet	231	A#3

Table 3.1 - Instrument Tones Analyzed by Stapleton and Bass

Based on the pitch of each tone, an estimate of the note played (based on equal-tempered tuning) is given in the third column of the table. Stapleton and Bass make no reference to the dynamic level at which each tone was played. Note that there are actually only thirteen instruments represented. Two of the tones are different renditions of violin: bowed and plucked. The renditions are distinct enough that they could be treated as different instruments.

After classification and some experimentation, the tones were grouped into three classes. The three classes are shown in Table 3.2. Classes 2 and 3 correspond completely to our intuition. Class 2 is comprised of brass instruments while Class 3 groups single and double reed woodwinds. (Originally, the classification procedure did not group the double reeds, oboe and bassoon, with Class 3. But experimentation showed that they could be included in Class 3 without increasing the number of basis functions required for quality synthesis beyond four.) Class 1, while not counter-intuitive, does surprise us with the grouping of wind, mallet and stringed instruments. Class 1 actually exhibits the highest inter-correlation between instruments. Four sample functions with a normalized sample period of 128 were taken from the sustain region of each tone to construct the sample function space for each class. The third column of Table 3.2 presents the number of basis functions required per instrument for quality synthesis of the tones. Generally, if an instrument requires N basis functions, this refers to the first N basis functions; however, in the case of the marimba and plucked violin, the best synthesis was achieved by using basis functions 1, 2, 3 and 5. The ordering of the basis functions is based on their average importance for the whole class. But for an individual tone within a class, this ordering may not strictly apply.

Class	Instrument	Number of Basis Functions for Synthesis
1	Diapason	2
	Flute	4
	Guitar	3
	Marimba	4
	Violin	4
	Plucked Violin	4
2	Trumpet	3
	French Horn	2
	Trombone	2
3	Alto Saxophone	3
	Tenor Saxophone	3
	Clarinet	4
	Oboe	4
	Bassoon	3

Table 3.2 - Instrument Classes Developed by Stapleton and Bass

Table 3.3 lists the basis function weights (eigenvalues) corresponding to basis functions 1 through 8 for each of the three classes. The retained energy expressed in percent is given as well. The retained energy for the first N basis functions is the sum of the first N weights expressed as a percentage. A reasonable measure of the compression achieved for a class is the number of basis functions required to achieve at least 99% energy retention. Class 1 requires 4 basis functions, Class 2 requires 3 and Class 3 requires 5. It is important to note a couple of points relating Tables 3.2 and 3.3. First, for the Class 1 instruments marimba and plucked violin, quality synthesis required the use of basis function 5 with a weight of .0053. Basis function 5 is not even

required to retain 99% of the energy in the sample function space, yet it is significant when considering the marimba and plucked violin tones. Second, for all of the Class 3 instruments, quality synthesis only required the first 4 basis functions, even though basis function 5 itself represents 4% of the sample function space energy. The first four basis functions only retain 95% of the the sample function space energy, yet are sufficient for quality synthesis. In summing up these points, we note that there is no fixed numerical criteria using basis function weights which we can use to determine how important an individual basis function is for any specific tone, or the number of basis functions required for quality synthesis.

Class 1		Class 2		Class 3	
Weight	Retained Energy [%]	Weight	Retained Energy [%]	Weight	Retained Energy [%]
.9259	92.59	.7792	77.92	.5139	51.39
.0403	96.62	.1660	94.52	.2870	80.08
.0135	97.98	.0483	99.35	.1009	90.17
.0109	99.07	.0033	99.68	.0505	95.22
.0053	99.60	.0020	99.88	.0407	99.29
.0016	99.76	.0008	99.95	.0023	99.52
.0010	99.87	.0002	99.97	.0022	99.74
.0005	99.92	.0001	99.98	.0009	99.83

Table 3.3 - Basis Function Weights and Retained Energy for Instrument Classes

Stapleton and Bass performed informal listening tests on their synthesized tones. Quality synthesis was defined as not necessarily indistinguishable from the original tone, but having the same quality and being recognizable as the original instrument type. For Class 1 instruments, all except the flute tone

were synthesized with high quality. The attacks were realistic for the guitar, marimba and plucked violin. The scraping sound of the bow for the violin was well preserved. The guitar and violin were slightly muffled due to the preliminary analysis low-pass filtering. The flute tone sounded very artificial. The basis functions could not reproduce the breathiness associated with a natural flute tone. Low amplitude white noise added to the synthesized flute tone was sufficient to restore the quality.

For Class 2 tones, the trombone and French horn were virtually indistinguishable from the original tones. The trumpet was slightly muffled. Attacks were well preserved. For Class 3, all the synthesized tones were of good quality.

One important comment about almost all the synthesized tones relates to a perceivable unnatural buzz during the decay. This buzz is due to the use of the frequency-rich basis functions in the decay. During the decay, typically the high frequency content of an instrument tone fades more quickly than the lower frequencies. The last few periods of a tone may be almost sinusoidal with only the fundamental remaining. With KL synthesis, all of the frequency content typical of the tone's sustain remains during the decay.

Stapleton and Bass also made a preliminary attempt to track individual instrument timbre with pitch. A natural instrument changes timbre with pitch. The degree of timbral change with pitch varies from instrument to instrument. Typically, resynthesizing pitches more than a few semitones away from the original tone pitch result in less natural sounding synthesis. A solution is to include more than one tone per instrument in the sample function space of a class. Stapleton and Bass tried using two pitches per instrument for Class 2. The resulting energy retention of the KL basis

functions was not as good, but listening tests showed that no additional basis functions were needed for quality synthesis compared to only using one pitch per instrument.

3.2. Results for One Instrument per Ensemble of Tones

In our work, we have attempted to further explore the use of the KL analysis/synthesis procedure, particularly in the area of tracking timbre variation for individual instruments. In contrast to the work of Stapleton and Bass, we have not attempted to combine different instruments into one class. Instead, we have analyzed different instruments as separate classes or ensembles of tones. For each instrument we have constructed several sample function spaces to analyze. The sample function spaces are presented in Table 3.4.

Instruments	Sample Function Space			
	1 Pitch	3 Pitches	3 Pitches x 2 Volumes	3 Pitches x 2 Styles
Alto Saxophone	•	•		
Nylon String Guitar	•	•		
Trumpet	•	•	•	
Alto Flute	•	•	•	
Oboe	•	•	•	
Cello Bowed	•	•		Bowed &
Cello Plucked	•	•		Plucked

Table 3.4 - Analyzed Sample Function Spaces

For each instrument, a sample function space of just 1 pitch and an ensemble of 3 pitches have been analyzed. In the case of 3 pitches, the notes

are approximately one octave apart, resulting in a total range of about two octaves. For the trumpet, alto flute and oboe a sample function space of 3 pitches with 2 dynamic levels (loud and soft) for each pitch has been analyzed. Finally, for cello, 3 pitches from both the bowed and plucked style have been combined into a single sample function space.

The sample function spaces have been constructed as follows. Eight single-period sample functions with normalized sample period 256 have been extracted from the sustain region of each tone. The nylon string guitar and plucked cello do not strictly have sustain regions, but a long slow decay. Their sample functions have been taken from over the duration of this gradual decay. The original sampled tones, sampled with 16-bit resolution at 44.1KHz, have been taken from the *Samplecell* CD-ROM produced by *Digidesign* and converted to a file format appropriate for the *Matlab*-based analysis/synthesis program.

The tones synthesized from the basis functions incorporate 100 linear segment approximations to the amplitude envelopes and frequency function. In a practical synthesis application, far less segments would be desirable, but for the purpose of comparing synthesized tones to the originals, maintaining a high resolution approximation is important. For each instrument, a common tone has been synthesized from each sample function space for the purpose of comparing realism as the sample function space is increased. We have generally restricted re-synthesis to only use the first four basis functions or less. No rigorous comparative listening tests have been performed with the synthesized tones. Qualitative comments regarding the original and re-synthesized tones are primarily the observations of the author, based on his own background knowledge of the instruments involved.

Alto Saxophone

Table 3.5 shows the compression achieved for each alto saxophone sample function space. For one pitch, only two basis functions are required for 99% energy retention. For three pitches, three basis functions are required. Re-synthesis of the F#4 tone was performed for both cases. The original tone exhibits a sharp attack followed by a quiet sustain. The re-synthesized tone constructed from two "1 Pitch" basis functions also exhibits a sharp attack. The texture of the sustain is quiet, but brighter than the original. Overall, the tone is realistic. The re-synthesized tone constructed from three "3 Pitch" basis functions has a less pronounced attack. As well, the sustained tone sounds slightly muffled; however the overall tone is still realistic and easily identifiable as an alto saxophone. For both re-synthesized tones, there is a noticeable buzz in the decay which is absent in the original tone.

1 Pitch F#4		3 Pitches G#3, F#4, E5	
Weight	Retained Energy [%]	Weight	Retained Energy [%]
.9875	98.75	.8634	86.34
.0116	99.91	.1061	96.95
.0008	99.99	.0215	99.09
.0001	100.00	.0042	99.52
.0000	100.00	.0028	99.80
.0000	100.00	.0011	99.91
.0000	100.00	.0002	99.94
.0000	100.00	.0002	99.96

Table 3.5 - Basis Function Weights and Retained Energy for Alto Saxophone Sample Function Spaces

Nylon String Guitar

Table 3.6 presents the compression results for each nylon string guitar sample function space. For one pitch, four basis functions are required for 99% energy retention. For three pitches, seven basis functions are required.

1 Pitch G3		3 Pitches A2, G3, E4	
Weight	Retained Energy [%]	Weight	Retained Energy [%]
.7277	72.77	.5995	59.95
.1532	88.09	.2470	84.65
.1021	98.30	.0641	91.07
.0084	99.14	.0347	94.53
.0049	99.63	.0272	97.25
.0027	99.90	.0127	98.52
.0008	99.98	.0053	99.05
.0002	100.00	.0042	99.47

Table 3.6 - Basis Function Weights and Retained Energy for Nylon Guitar Sample Function Spaces

The large number of basis functions required for just one tone is a result of the way a plucked string's timbre changes as the tone decays. Figure 3.1 presents the sample functions used in the "1 Pitch" sample function space. Clearly there is significant variation between the sample functions.

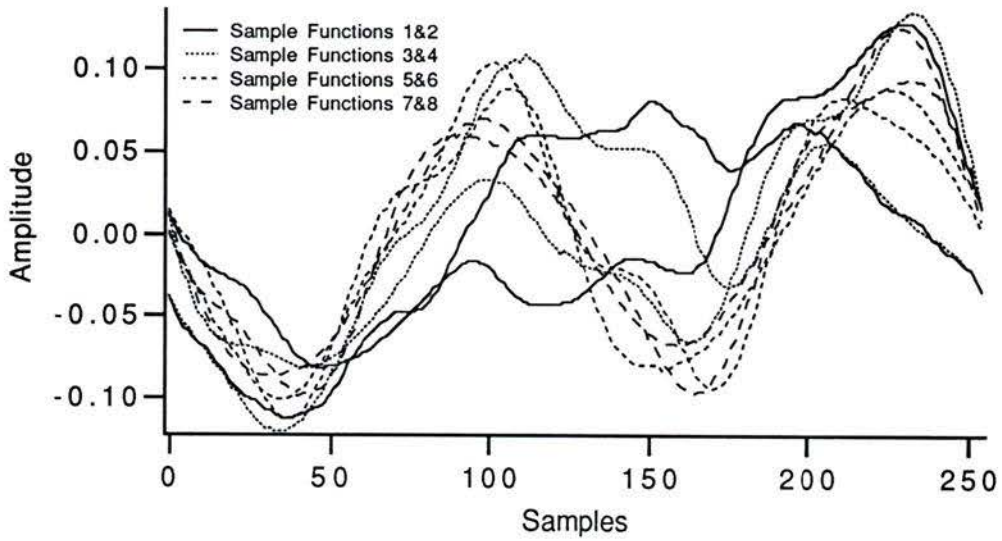

Figure 3.1 - Eight Phase-Aligned Sample Functions extracted from Single Nylon String Guitar Tone

Re-synthesis of the G3 tone was performed for both sample function spaces. The original tone exhibits a strong plucked attack followed by an even decay. The re-synthesized tone constructed from the "1 Pitch" basis functions has a duller, but still realistic, plucking sound. A barely noticeable amplitude warble can be heard at the beginning of the decay. The re-synthesized tone constructed from the "3 Pitch" basis functions has an even duller, somewhat unrealistic, pluck. The tone is rendered completely unrealistic by a much stronger amplitude warble or modulation in the early decay. This amplitude warble can be attributed to the strong negative-positive oscillations of the third basis function's amplitude envelope, as illustrated in Figure 3.2. The amplitude envelopes of the second, third and fourth "1 Pitch" basis functions all exhibit slight negative-positive oscillations, accounting for the slight warble detected.

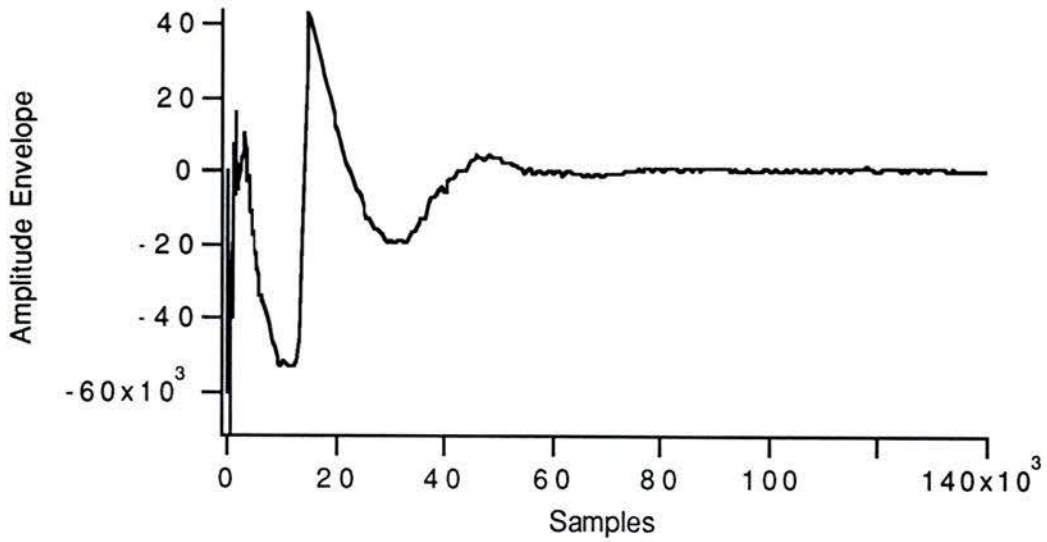

Figure 3.2 - Amplitude Envelope of Nylon String Guitar Basis Function Exhibiting Strong Negative-Positive Oscillations

Trumpet

Table 3.7 shows the compression achieved for each trumpet sample function space. For one pitch, only two basis functions are required for 99% energy retention. For three pitches, four basis functions are required. For three pitches and two volumes, six basis functions are required. Re-synthesis of the G4 tone was performed for all three cases. The "1 Pitch" tone was constructed from two basis functions; the other two cases used four basis functions. The original tone is bright and even. The re-synthesized tones for all three cases are virtually indistinguishable from each other. They exhibit a realistic though less pronounced attack than the original tone. The sustain of the re-synthesized tones sounds identical to the original. For all three re-synthesized tones, similar to the alto saxophone tones, there is a noticeable buzz in the decay which is absent in the original tone.

1 Pitch G4		3 Pitches A3, G4, F5		3 Pitches x 2 Volumes A3, G4, F5 *p, f*	
Weight	Retained Energy [%]	Weight	Retained Energy [%]	Weight	Retained Energy [%]
.9847	98.47	.8336	83.36	.8103	81.03
.0118	99.65	.1257	95.93	.1255	93.59
.0023	99.88	.0280	98.73	.0303	96.61
.0007	99.94	.0042	99.15	.0132	97.93
.0003	99.97	.0031	99.46	.0098	98.91
.0001	99.99	.0019	99.65	.0028	99.19
.0001	99.99	.0008	99.73	.0019	99.38
.0001	100.00	.0007	99.81	.0017	99.54

Table 3.7 - Basis Function Weights and Retained Energy for Trumpet Sample Function Spaces

Alto Flute

Table 3.8 provides the compression results for each alto flute sample function space. For one pitch, only two basis functions are required for 99% energy retention. For both three pitches and three pitches x two volumes, five basis functions are required. Volume level does not seem to be a significant parameter in varying the alto flute timbre, as it does not significantly increase the dimension of the sample function space. Re-synthesis of the F4 tone was performed for all three cases. The "1 Pitch" tone was constructed from two basis functions; the other two cases used four basis functions. The original tone exhibits a substantial "breathiness" which we expect from a member of the flute family. The three re-synthesized tones, as with the trumpet, are virtually indistinguishable from each other; however, they are quite unlike

the original tone. The synthesized tones lack the "breathy" quality of the original. Instead they sound like a cross between flute and saxophone.

1 Pitch F4		3 Pitches G3, F4, D#5		3 Pitches x 2 Volumes G3, F4, D#5 *p, f*	
Weight	Retained Energy [%]	Weight	Retained Energy [%]	Weight	Retained Energy [%]
.9803	98.03	.5969	59.69	.7077	70.77
.0156	99.60	.2406	83.75	.1861	89.38
.0023	99.83	.1289	96.64	.0713	96.50
.0008	99.91	.0219	98.83	.0170	98.20
.0005	99.96	.0059	99.42	.0081	99.01
.0002	99.98	.0024	99.67	.0045	99.45
.0001	99.99	.0012	99.79	.0017	99.62
.0001	100.00	.0008	99.87	.0011	99.74

Table 3.8 - Basis Function Weights and Retained Energy for Alto Flute Sample Function Spaces

Oboe

Table 3.9 shows the compression achieved for each oboe sample function space. For one pitch, only one basis functions is required for 99% energy retention. For three pitches, three basis functions are required. For three pitches and two volumes, six basis functions are required. It is interesting to note in this last case that even the sixth basis function is somewhat significant, representing 3.5% of the sample function space energy. Curiously, the number of significant basis functions is identical to the number of original tones contributing to the sample function space for each case. The sample functions from one tone are highly correlated with one another while correlation between tones at different pitches and/or volume is much less.

This can be seen in Figure 3.3, where the phase-aligned sample functions for the third case are shown. Note that six distinct waveforms are represented. The waveforms do not seem to align well either.

1 Pitch D5		3 Pitches D4, C5, A#5		3 Pitches x 2 Volumes D4, C5, A#5 *p, f*	
Weight	Retained Energy [%]	Weight	Retained Energy [%]	Weight	Retained Energy [%]
.9976	99.76	.6336	63.36	.5116	51.16
.0015	99.91	.2247	85.83	.1935	70.51
.0004	99.95	.1392	99.76	.1260	83.11
.0002	99.97	.0008	99.84	.0769	90.80
.0001	99.98	.0007	99.90	.0557	96.37
.0001	99.99	.0003	99.93	.0345	99.82
.0001	100.00	.0002	99.95	.0006	99.88
.0000	100.00	.0002	99.97	.0004	99.91

Table 3.9 - Basis Function Weights and Retained Energy for Oboe Sample Function Spaces

Re-synthesis of the C5 tone was performed for all three cases. The "1 Pitch" tone was constructed from one basis function. The "3 Pitches" tone used three basis functions. A tone for the "3 Pitches x 2 Volumes" case was first constructed from six basis functions. The re-synthesized tones for all three cases are virtually indistinguishable from each other and from the original. A four basis function tone was also constructed for the third case. This tone was not as bright as the others, but still seemed realistic.

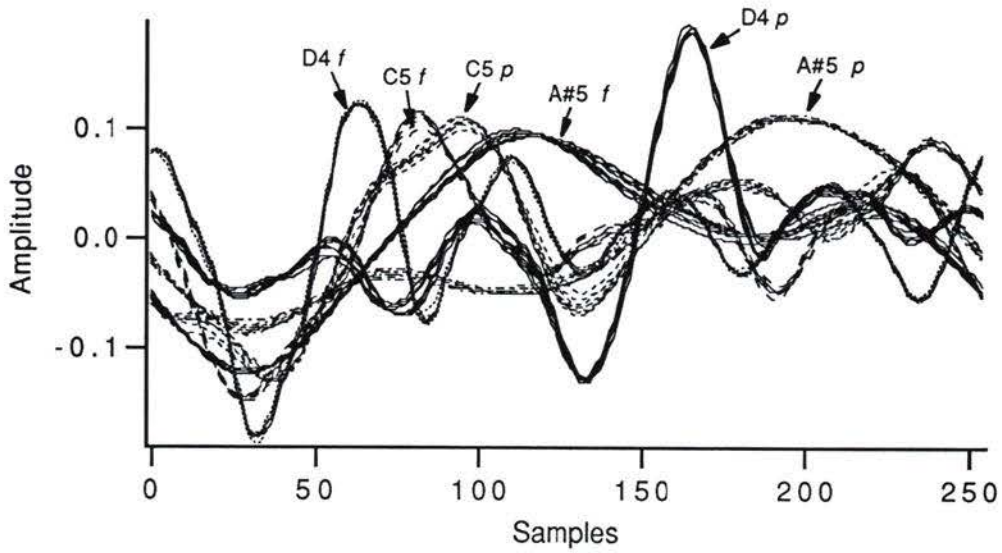

Figure 3.3 - Phase-Aligned Sample Function Space for Oboe with "3 Pitches x 2 Volumes"

Cello Bowed

Compression for each bowed cello sample function space is presented in Table 3.10. For one pitch, two basis functions are required for 99% energy retention. For 3 pitches, five basis functions are required. For three pitches of both bowed and plucked cello, eight basis functions are required. Re-synthesis of the D#3 tone was performed for all three cases. The "1 Pitch" tone was constructed from two basis functions, the other two tones from four basis functions. The original tone exhibits a strong buzz from the bow against the strings. The "1 Pitch" tone has only a slight sound of bowing, but seems quite realistic. The author actually prefers this synthesized tone to the original tone; the recording of the original tone seems to have unnaturally accentuated the bowing sound. The "3 Pitch" tone is very close the the "1 Pitch" tone, with a slightly lesser bowing sound. The tone for the bowed/plucked sample

function space is still realistic, but sounds muffled compared to the first two cases.

1 Pitch D#3		3 Pitches C2, D#3, C#4		3 Pitches x 2 Styles C2, D#3, C#4 Bowed C2, D3, C4 Plucked	
Weight	Retained Energy [%]	Weight	Retained Energy [%]	Weight	Retained Energy [%]
.9857	98.57	.6266	62.66	.5757	57.57
.0071	99.28	.2589	88.55	.2248	80.05
.0054	99.82	.0830	96.85	.0910	89.15
.0007	99.89	.0177	98.62	.0451	93.67
.0004	99.93	.0039	99.01	.0266	96.32
.0003	99.97	.0029	99.30	.0152	97.84
.0002	99.99	.0021	99.51	.0079	98.63
.0001	100.00	.0019	99.69	.0038	99.01

Table 3.10 - Basis Function Weights and Retained Energy for Bowed Cello Sample Function Spaces

Cello Plucked

Compression for each plucked cello sample function space is presented in Table 3.11. For one pitch and three pitches, five basis functions are required for 99% energy retention. The bowed/plucked sample function space data is reproduced from Table 3.10 for convenience. The reason for the low compression in the one pitch case is the same as for the nylon string guitar, also a plucked instrument. Re-synthesis of the D3 tone was performed for all three cases using four basis functions. The original tone exhibits a definite plucked attack followed by an even decay. For the "1 Pitch" tone, the plucking sound is muted, and a slight amplitude warble renders the tone unrealistic.

The "3 Pitches" tone is the most realistic, having a lesser amplitude warble. The tone generated by basis function from the bowed/plucked case exhibits a bowed attack instead of plucked. The inclusion of bowed tones in the sample function space has ruined the plucked attack.

1 Pitch D3		3 Pitches C2, D3, C4		3 Pitches x 2 Styles C2, D#3, C#4 Bowed C2, D3, C4 Plucked	
Weight	Retained Energy [%]	Weight	Retained Energy [%]	Weight	Retained Energy [%]
.7763	77.63	.5779	57.79	.5757	57.57
.1650	94.13	.3165	89.43	.2248	80.05
.0304	97.16	.0482	94.26	.0910	89.15
.0156	98.72	.0358	97.84	.0451	93.67
.0062	99.34	.0127	99.11	.0266	96.32
.0043	99.77	.0037	99.48	.0152	97.84
.0020	99.96	.0026	99.74	.0079	98.63
.0004	100.00	.0016	99.90	.0038	99.01

Table 3.11 - Basis Function Weights and Retained Energy for Plucked Cello Sample Function Spaces

3.3. Discussion of Results

The majority of the instrument tones we chose to analyze and re-synthesize work well with the KL basis function technique. The alto saxophone, trumpet and oboe tones can be analyzed alone or grouped with multiple pitches/volumes and produce satisfactory synthesized tones for all cases. The oboe exhibits significant timbre variation with both pitch and volume resulting in the need to retain six basis functions for a sample

function space consisting of 3 pitches x 2 volumes. This exceeds the desired four basis function limit for re-synthesis. The nylon string guitar and bowed cello exhibit the best re-synthesis results when only one tone contributes to the sample function space. The plucked cello re-synthesizes poorly from the automatically determined amplitude envelopes, due to positive/negative oscillations of at least one of the significant basis functions. The nylon string guitar tones also exhibit this problem. It is worth noting that these two instruments are the only plucked string instruments of those analyzed as well as being the only instruments to manifest problems with amplitude envelope oscillations. Finally, the alto flute does not synthesize well with periodic KL basis functions due to their inability to produce the noise-like component of breathiness. Note that the best synthesized bowed cello tone, while realistic, did not preserve the strong bowing sound of the original tone. The bowing sound is also noise-like.

As expected, expanding the sample function space from one pitch to multiple pitches and possibly volumes typically results in the need to use more basis functions for quality synthesis. In the case of the oboe, the expansion in the number of significant basis functions corresponds to the number of tones in the space. For all of the instruments except plucked strings, if only one tone is analyzed, the number of significant basis functions is restricted to only two. The plucked string tones exhibit the worst compression with three or four significant basis functions. Generally, the best quality re-synthesis results from using only one analyzed tone. Inclusion of multiple pitches and volumes in a sample function space does not typically degrade the quality of re-synthesis to the point at which tones are no longer

realistic. Combining bowed and plucked cello is not as successful. It would be better to treat these as separate instruments.

Our initial intention in analyzing multiple pitches and possibly volumes in a single sample function space was to derive a small set of common KL basis function oscillators which could be used to realistically synthesize all pitches and dynamic levels of an instrument. Examination of the results indicates a potentially better approach to using KL basis function oscillators for synthesizing the full pitch and dynamic range of an instrument. Instead of trying to map the whole range of an instrument into a single set of oscillators, perhaps it would be better to implement a KL-based synthesizer similar to a sample-playback synthesizer. Separate KL basis functions would be computed for each pitch and volume level to be exactly represented. The pitches and volume levels exactly represented are only a sparse sampling of the full instrument range. Pitch and volume are scaled for tones not exactly represented.

The advantages that such a KL-based synthesizer would have over sample-playback synthesizers is a significant reduction in the amount of memory required to synthesize tones, as well as greater variation in synthesized tones due to the ability to change oscillator weightings (if two or more oscillators are used). The advantages of using separate KL basis functions for each represented tone instead of a common set is typically a reduction in the number of oscillators required per synthesized tone as well as a general improvement in the quality of synthesized tones, at least for those notes exactly represented. We expect that, as with sample-playback synthesizers, tones near those exactly represented will also be synthesized well. The spacing (in pitch and volume) between represented tones would

depend on the rate at which timbre changes for any given instrument. As an example from our results, we would probably use a finer sampling of oboe tones than of alto saxophone tones.

Some general comments can be made regarding deficiencies in the KL basis function technique which are evident in the results. Attacks typically are not as pronounced and bright. There is a general problem with noisy decays due to the use of harmonically rich basis functions. A problem with oscillating amplitude envelopes has been encountered with plucked string instruments. Finally, the KL basis functions cannot re-synthesize noise-like components of tones such as breathiness and strong bowing sounds. Our findings regarding decay buzz and an inability to synthesize noise concur with those of Stapleton and Bass; however Stapleton and Bass make no mention of less pronounced attacks or a problem with oscillating amplitude envelopes.

In summing up our results, we feel that KL-based synthesis still holds promise as a cost-effective synthesis technique in terms of both computational complexity and memory requirements. Recommendations for overcoming the deficiencies inherent in the technique are given in the following chapter.

CHAPTER 4
SUGGESTIONS FOR FUTURE WORK

The current KL analysis/synthesis scheme could be improved in a number of ways. Typically, synthesis of the sustained portions of harmonic instrument tones is of high quality. Deficiencies exist in the synthesis of both the attack and decay portions of tones. In its present form, KL analysis/synthesis is not suitable for application to non-harmonic tones. Finally, the near-periodic repetition of wavetables inherent in KL synthesis cannot reproduce noiselike components in instrument tones.

Most of these deficiencies have been encountered with other analysis and synthesis techniques. A number of schemes have been developed to improve on these deficiencies in the context of the other analysis/synthesis techniques. These improvements could also be added to the basic KL analysis/synthesis procedure. The first three sections of this chapter will in turn deal with the analysis/synthesis of attack and decay, non-harmonic tones and finally noiselike components. The final section will discuss the design of a synthesizer incorporating the basic KL analysis/synthesis scheme as well as the suggested improvements.

4.1. Improving Attack and Decay Synthesis

The KL analysis scheme we have presented does not include in its analysis space sample functions from either the attack or decay portions of a tone. We have chosen to omit sample functions from the attack and decay because they are less periodic than sample functions from the sustain. This is especially

true for instruments with quick attacks and/or decays. For plucked string instruments, we have treated the long decay as the sustain region; however, the variation in timbre over a plucked string decay is significant, resulting in significantly less compression in the KL basis functions. One approach to improving attack and decay synthesis would be to include sample functions from both regions; however, our experience with the plucked string decays indicates that this would significantly increase the number of KL basis functions required for realistic synthesis. As well, we expect that the quality of synthesis in the sustain region will decrease if basis functions derived from sustain as well as attack and decay sample functions are used.

Another approach could be to derive separate basis functions for the attack, sustain and decay regions of the tones. During re-synthesis, care must be taken to smoothly change wavetables between regions. This requires aligning the phase as wavetables are switched and possibly performing a cross-fade as the wavetables of one region are faded out and the wavetables of the next region faded in. During cross-fades, this would require double the number of wavetable oscillators running.

One approach to improving the realism of attack synthesis has been mentioned in the context of traditional Fourier-based additive synthesis [13]. Fourier-based additive synthesis has trouble realistically synthesizing attacks due to their typically transient and noisy characteristics. KL-based synthesis has a similar drawback due to the use of periodic wavetables. The suggested improvement is to splice two or three periods of the original tone's early attack into the beginning of the synthesized tone. This does not require too much extra data to be stored for the tone. For the splice to be rendered inaudible, the wavetable synthesis should commence in phase with the end

of the attack sample while cross-fading from the sampled to synthesized tone. This should not be difficult to achieve.

If improvements are made to analysis and re-synthesis of the noiselike components of a tone, this may be sufficient to render attacks sufficiently realistic, since it is often the attack noise which is missing in KL-based synthesis attacks. A final possibility is to alter the automatically generated attack envelopes (from the analysis procedure) in such a manner as to create more pronounced attacks.

The typical problem with unrealistic decays is an audible buzzing due to the use of harmonically rich basis functions all the way to the end of a tone. This buzzing could be masked by using a time-varying low-pass filter during the decay synthesis. At the beginning of the decay, the filter would have a flat response, not yet attenuating high frequencies. By the end of the tone, the filter's cutoff frequency would be just above the fundamental frequency of the tone. This produces the effect of higher frequencies dying away more quickly than lower frequencies during the decay. Experiments need to be performed to determine the effectiveness of this scheme, and what order filter is sufficient to implement it. Obviously, the lower the necessary filter order, the better.

4.2. Synthesizing Non-Harmonic Tones

Due to the common phase index used in cycling through the basis functions, KL synthesis cannot reproduce non-harmonic tones. One approach has been suggested for dealing with non-harmonic tones which, while not periodic, do exhibit a steady-state harmonic spectrum [38]. For such a tone, while the frequency peaks in the spectrum are not harmonically related, the

peaks do not vary significantly in frequency over the tone's duration. Such a tone, while not periodic with respect to its pitch or fundamental frequency (the lowest significant spectral peak), can be thought of as having a much lower imaginary fundamental frequency. This imaginary fundamental frequency is simply the greatest common divisor of the frequencies of all the significant spectral peaks. One would first have to perform spectral analysis of a non-harmonic tone to determine an imaginary fundamental frequency. An imaginary fundamental should probably not be less than several Hz (to avoid excessively long sample functions). Achieving such a high imaginary fundamental may require that it not exactly be a common divisor of all the spectral peaks, but is close to an exact divisor for each of the peaks.

We would use the sample period associated with the imaginary fundamental as the length of sample functions to extract from the non-harmonic tone. The normalized sample period (for analyzing multiple tones at different pitches) should be increased proportionately. The KL analysis procedure would be applied to these longer sample functions, and synthesis performed with the longer KL basis function wavetables. The wavetable memory requirements would be several times those for harmonic KL synthesis. All of the above is merely speculative, and needs to be researched to determine if a reasonable number of non-harmonic tones could be analyzed and synthesized using the general technique described.

4.3. Synthesizing Noiselike Components of Tones

Similar to dealing with the attack portion of tones, both KL synthesis and Fourier-based additive synthesis cannot reproduce noiselike components. This deficiency in Fourier-based synthesis has been dealt with in detail by

Serra and Smith [14]. Serra and Smith separated a musical tone into two components - a deterministic part and a stochastic part. The deterministic part is modeled by the sum of time-varying sinusoids which may or may not be harmonically related. The stochastic part is modeled as the residue between the synthesized and original tones in the frequency domain. The analysis procedure consists of first computing the deterministic component of the synthesis, re-synthesizing the signal from time-varying sinusoidal oscillators, and subtracting the magnitude spectrum of the re-synthesized tone from the magnitude spectrum of the original tone to determine a frequency envelope for the noise spectrum. Synthesis of the stochastic component can be implemented by passing white noise through a time-varying filter which tracks the shape of the noise spectrum. An alternative implementation is to combine the frequency envelope of the noise with random phase information as the inputs to an Inverse Fast Fourier Transform (IFFT), using the overlap-add method to produce a smooth time-domain output.

The modeling of a tone's stochastic or noiselike component described above can be directly applied to augment the KL analysis procedure. A block diagram of the augmented analysis procedure is given in Figure 4.1. It would be useful to investigate how simple the approximations to the noise spectrum's envelope can be made in order to facilitate low computational cost in re-synthesis. Ideally, re-synthesis would only require passing pseudo-random white noise through a low-order filter to adequately reproduce the noiselike component of a tone.

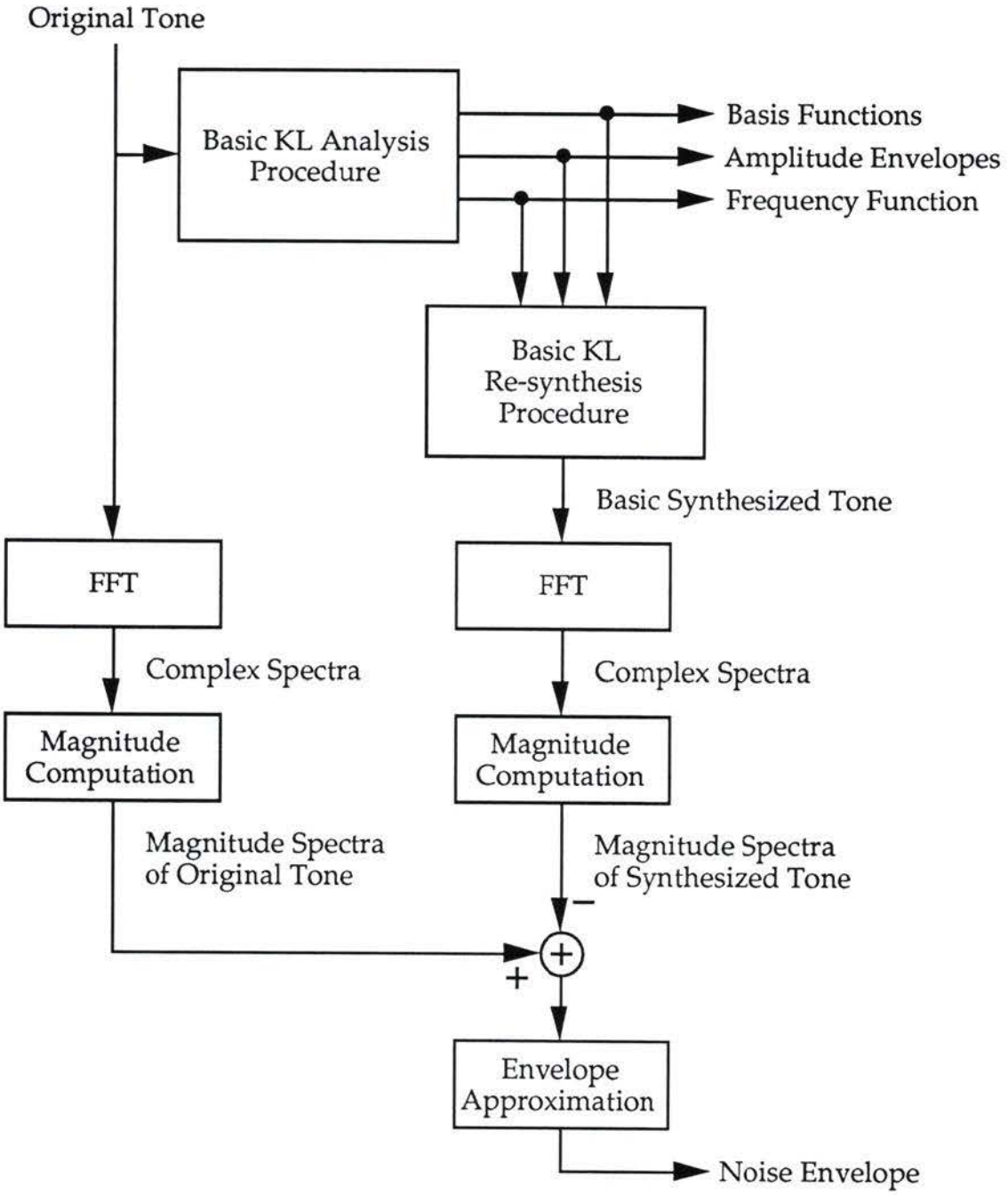

Figure 4.1 - Augmenting the KL Analysis Procedure to Model the Stochastic Component of a Tone

4.4. Designing a KL-based Synthesizer

It would be useful to implement a real-time KL-based synthesizer. This would allow much quicker turn-around for auditioning re-synthesized tones. A suggested block diagram for a basic KL synthesizer incorporating four KL basis function oscillators is presented in Figure 4.2. The block diagram implements synthesis of a single tone at a time. The synthesizer consists of a phase generator block, oscillator blocks, envelope generator blocks, memory and simple multipliers and an adder. The phase generator block updates the phase index sample by sample based on the frequency function stored in memory. The phase generator block outputs a real number restricted to the range $0.0 \leq$ phase index $< N$, where N is the length of the oscillator basis functions or wavetables stored in memory. The oscillators use the phase index as follows. The integer part of the phase index is used to address two adjacent wavetable samples, and the fractional part used to compute a linearly interpolated oscillator output from the two adjacent samples. A linearly interpolating oscillator improves the signal-to-noise ratio of the output. The oscillator outputs are scaled by the gain factors of their associated envelope generators, then summed to form the synthesized tone. Both the phase and envelope generators could incorporate low-frequency oscillators to implement tonal modifications such as vibrato and tremolo. Not shown in the block diagram are control signals for starting, updating and stopping a tone.

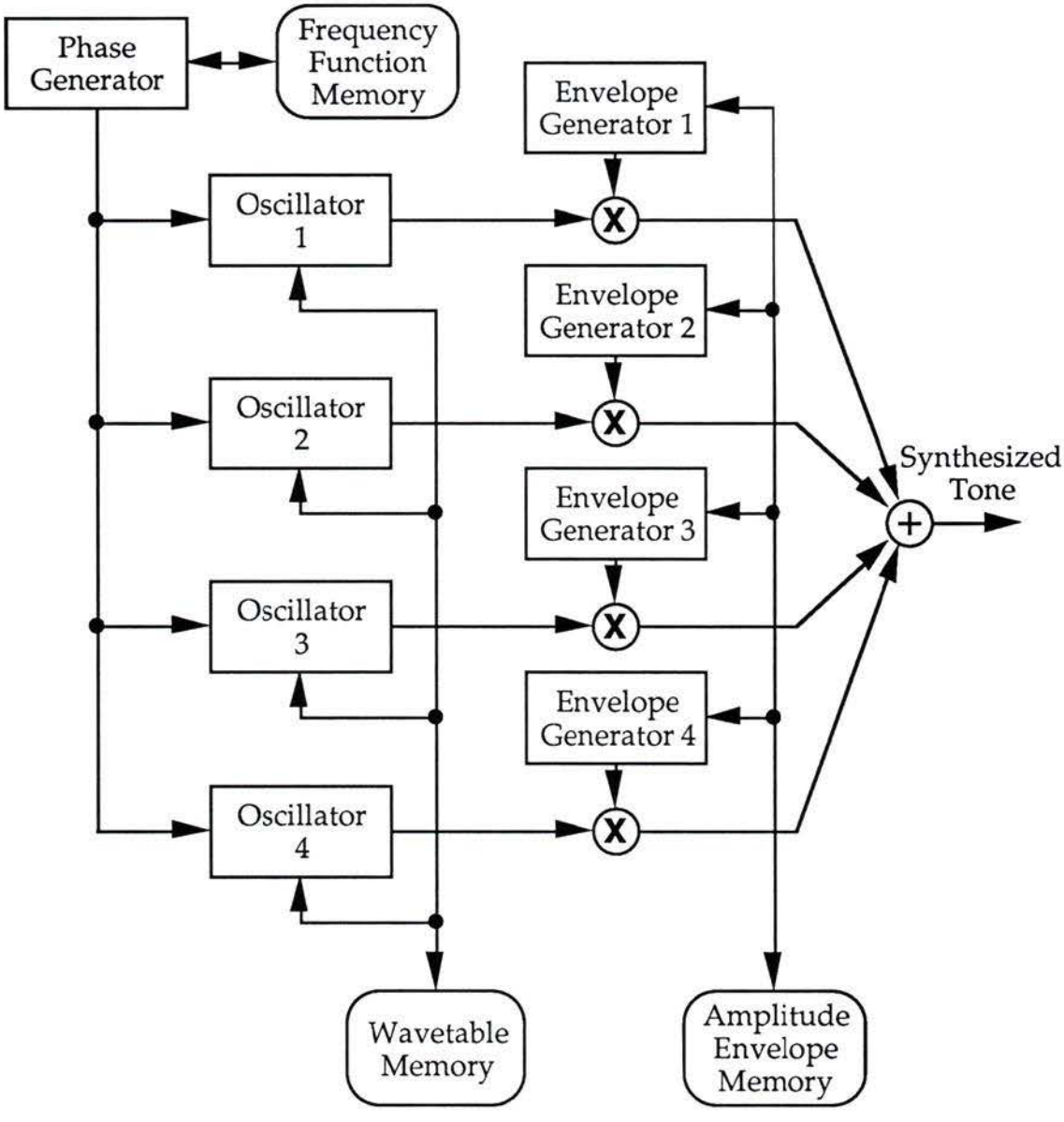

Figure 4.2 - Block Diagram of Basic KL Synthesizer Incorporating Four Wavetable Oscillators

The block diagram can be extended to provide multi-tone synthesis. The block diagram could be implemented either in custom hardware or a general-purpose digital signal processor or microprocessor. A useful example of

implementing the phase generator and wavetable oscillators on a Motorola DSP56000 digital signal processor is found in [39].

Several suggestions for improving the KL analysis/synthesis procedure have already been made in this chapter. A block diagram of a KL synthesizer incorporating the basic KL synthesis engine of Figure 4.2 and two of the suggested improvements is presented in Figure 4.3. We suggest that sufficient synthesis quality may be achieved by the inclusion of modeling the noiselike tone components and low-passing the decay portion of tones. The decay low-pass would normally only cut in after the "Stop Note" control for sustained tones. For gradually decaying tones, the low-pass could start right after the attack. If sufficiently low-order filters are used for both the decay and noise envelope, the computational cost of synthesis should not be prohibitive. Techniques for implementing time-varying recursive digital filters can be found in [40].

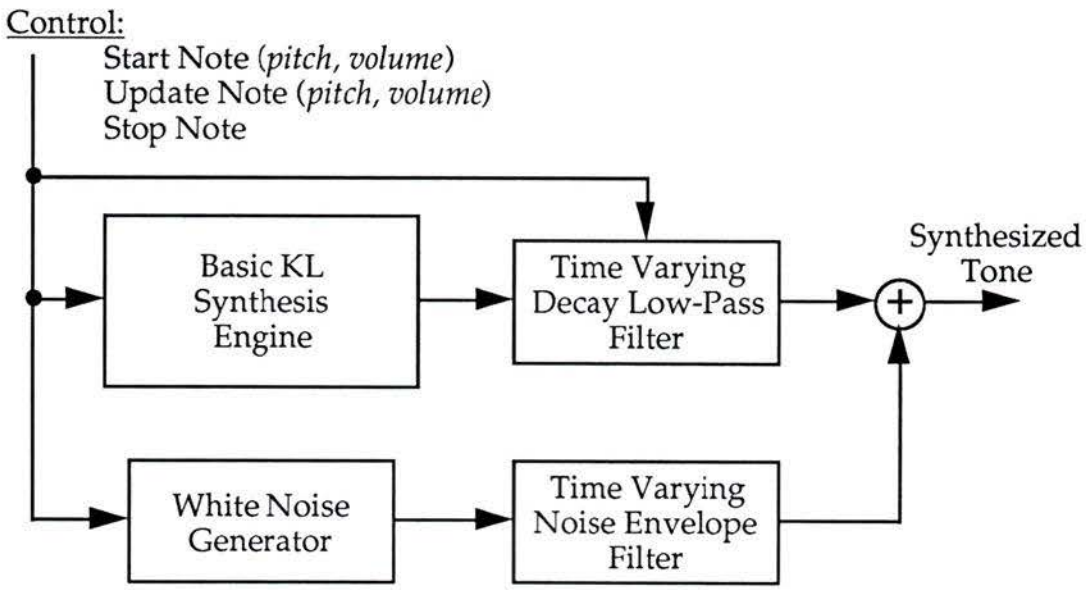

Figure 4.3 - Block Diagram of Proposed Improved KL Synthesizer

CHAPTER 5
SUMMARY AND CONCLUSIONS

The analysis and synthesis of musical tones based on the Karhunen-Loève Transform is just one of many analysis/synthesis techniques which have been developed in recent years. Each technique has its own advantages as well as drawbacks. Compared to Fourier-based additive synthesis, KL-based synthesis is much less computationally expensive. Compared to sample playback synthesis, KL-based synthesis requires less data memory. Compared to Frequency Modulation synthesis, KL-based synthesis has a well-defined analysis procedure for determining synthesis parameters. The fundamental drawback of KL-based analysis/synthesis is the requirement for modeling a musical tone as near-periodic.

In this thesis, we have reviewed the original work of Stapleton and Bass on KL-based analysis/synthesis. We have made small refinements and extensions to the analysis procedure as well as further formalizing the overall procedure. We have applied our analysis procedure to individual instruments in order to determine how well a small number (four) of KL basis function oscillators can track the variations in an instrument's timbre over pitch, dynamic level and even playing style. For some instruments, such as trumpet, the tracking is excellent. But other instruments exhibiting greater timbral variation, such as oboe, are not as well tracked with a small number of KL basis functions. Mixing two playing styles, such as bowed and plucked string, produced unsatisfactory results.

Based on our results, we suggest that the best use of KL-based analysis/synthesis is to separately analyze different tones for one instrument and set up synthesis similar to a sample playback synthesizer, which uses several tone samples for any one instrument. Such a KL-based synthesizer would require less memory but more computational power than a sample playback synthesizer. The use of at least two KL basis function oscillators in the synthesis affords greater timbral variation of synthesized tones than for sample playback. It is also possible to analyze tones from two different instruments and set up synthesis of hybrid instruments by varying the amplitude envelopes of the KL basis function oscillators.

In Chapter 4 we have made suggestions for future work. We recommend that noise analysis and modeling be added to the existing analysis procedure. This would expand the number of instruments which could be realistically synthesized. It would also be useful to attempt adapting the analysis procedure for modeling at least some subset of non-harmonic tones, further increasing the procedure's generality. Finally, implementing a real-time synthesizer based on the block diagrams presented in Section 4 of Chapter 4 would be necessary for full exploration of the range of useful tones that can be synthesized using the KL-based techniques.

In summation, we feel that analysis and synthesis based on the Karhunen-Loève Transform is a promising technique which warrants further development and enhancements. With the existing analysis/synthesis procedure, we have achieved quality re-synthesis of instrument tones which are near-periodic and have minimal noise components. If the improvements suggested are successful, then it will be possible to analyze and synthesize a much wider range of instrument tones with similar quality. Finally, the

complexity of our proposed KL-based synthesizer is comparable to existing commercially available synthesizers. Our proposed synthesizer would use between one and four oscillators, a noise-generator and two time-varying, low-order, recursive filters per tone. Commercially available sample playback synthesizers use only one oscillator per tone, but at the expense of larger memory requirements than for KL-based synthesis. Other commercially available synthesizers typically use between two and six oscillators per tone. These synthesizers typically have memory requirements comparable to or somewhat less than KL-based synthesis. The quality of synthesis for commercially available non-sample playback synthesizers based on their ability to re-synthesize existing instrument tones is often less than desired. Both the quality of synthesized tones and the reasonable computational/ memory costs for synthesis make the KL-based analysis/synthesis techniques worth considering.

LIST OF REFERENCES

1. Y. Menuhin and C.W. Davis. *The Music of Man*, Methuen, 1979.

2. W.A. Schloss. "Recent Advances in the Coupling of the Language Max with the Mathews/Boie Radio Drum," *Proceedings of the International Computer Music Conference*, 1990.

3. *MIDI 1.0 Specification*, International MIDI Association, 1983.

4. *Current Directions in Computer Music Research*, Ed. M.V. Mathews and J.R. Pierce, The MIT Press, 1989.

5. F.R. Moore. *Elements of Computer Music*, Prentice Hall, 1990.

6. E. Lindemann, F. Dechelle, B. Smith, and M. Starkier. "The Architecture of the IRCAM Musical Workstation," *Computer Music Journal*, Vol. 15, No. 3, pp. 41-49, 1991.

7. G. De Poli. "A Tutorial on Digital Sound Synthesis Techniques," *The Music Machine - Selected Readings from Computer Music Journal*, The MIT Press, pp. 429-447, 1989.

8. J.W. Gordon. "System Architectures for Computer Music," *Computing Surveys*, Vol. 17, No. 2, pp. 191-233, June 1985.

9. J.O. Smith. "Viewpoints on the History of Digital Synthesis," *Proceedings of the International Computer Music Conference*, pp. 1-10, 1991.

10. G.J. Sandell. "A Library of Orchestral Instrument Spectra," *Proceedings of the International Computer Music Conference*, pp. 98-101, 1991.

11. R. Kronland-Martinet. "The Wavelet Transform for Analysis, Synthesis, and Processing of Speech and Music Sounds," *Computer Music Journal*, Vol. 12, No. 4, pp. 11-20, 1988.

12. R. Plomp. *Aspects of Tone Sensation*, Academic Press, 1976.

13. J.O. Smith and X. Serra. "PARSHL: An Analysis/Synthesis Program for Non-Harmonic Sounds Based on a Sinusoidal Representation," *Proceedings of the International Computer Music Conference*, pp. 290-297, 1987.

14. X.J. Serra and J.O. Smith. "Spectral Modeling Synthesis: A Sound Analysis/Synthesis System Based on a Deterministic plus Stochastic Decomposition," *Computer Music Journal*, Vol. 14, No. 4, pp. 12-24, 1990.

15. E.B. George and M.J.T. Smith. "An Analysis-by-Synthesis Approach to Sinusoidal Modeling Applied to the Analysis and Synthesis of Musical Tones," *Proceedings of the International Computer Music Conference*, pp. 356-359, 1991.

16. S.C. Bass and T.W. Goeddel. "The Efficient Digital Implementation of Subtractive Music Synthesis," *IEEE MICRO*, Vol. 1, No. 3, pp. 25-37, August 1981.

17. T.W. Goeddel and S.C. Bass. "High Quality Synthesis of Musical Voices in Discrete Time," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 32, No. 3, pp. 623-633, June 1984.

18. P.R. Cook. "TBone: An Interactive WaveGuide Brass Instrument Synthesis Workbench for the NeXT Machine," *Proceedings of the International Computer Music Conference*, pp. 297-299, 1991.

19. S.E. Hirschman, P.R. Cook, and J.O. Smith. "Digital Waveguide Modelling of Reed Woodwinds: An Interactive Development," *Proceedings of the International Computer Music Conference*, pp. 300-303, 1991.

20. J.O. Smith. "Waveguide Simulation of Non-Cylindrical Acoustic Tubes," *Proceedings of the International Computer Music Conference*, pp. 304-307, 1991.

21. J.M. Chowning. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation," *Journal of the Audio Engineering Society*, Vol. 21, No. 7, pp. 526-534, September 1973.

22. R.G. Payne. "A Microcomputer Based Analysis/Resynthesis Scheme for Processing Sampled Sounds using FM," *Proceedings of the International Computer Music Conference*, pp. 282-289, 1987.

23. H. Chamberlin. *Musical Applications of Microprocessors*, Hayden Book Company, 2nd Edition, 1985.

24. W.A. Schloss. *On the Automatic Transcription of Percussive Music - From Acoustic Signal to High-Level Analysis*, Ph.D. Dissertation, Report No. STAN-M-27, Stanford University, May 1985.

25. L. Torres-Urgell and R.L. Kirlin. "Adaptive Image Compression using Karhunen-Loève Transform," *Signal Processing*, Vol. 21, No. 4, pp. 303-313, December 1990.

26. A.S. Spanias, S.B. Jonsson, and S.D. Stearns. "Transform Methods for Seismic Data Compression," *IEEE Transactions on Geoscience and Remote Sensing*, Vol. 29, No. 3, pp. 407-416, May 1991.

27. S.M.S. Jalaleddine, C.G. Hutchens, R.D. Strattan, and W.A. Coberly. "ECG Data Compression Techniques - A Unified Approach," *IEEE Transactions on Biomedical Engineering*, Vol. 37, No. 4, pp. 329-343, April 1990.

28. S. Haykin. *Modern Filters*, Macmillan Publishing Company, 1989.

29. J.C. Stapleton. *Karhunen-Loève Based Additive Synthesis of Musical Tones*, Ph.D. Dissertation, Purdue University, December 1985.

30. J.T. Tou and R.C. Gonzalez. *Pattern Recognition Principles*, Addison-Wesley Publishing Company, 1974.

31. J.C. Stapleton and S.C. Bass. "Synthesis of Musical Tones Based on the Karhunen-Loève Transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 36, No. 3, pp. 305-319, March 1988.

32. R.E. Crochiere and L.R. Rabiner. "Interpolation and Decimation of Digital Signals - A Tutorial Review," *Proceedings of the IEEE*, Vol. 69, No. 3, pp. 300-331, March 1981.

33. D.S. Watkins. *Fundamentals of Matrix Computations*, John Wiley & Sons, 1991.

34. *Matlab User's Guide*, The MathWorks, Inc., 1990.

35. R.L. Burden and J.D. Faires. *Numerical Analysis*, Prindle, Weber & Schmidt, 3rd Edition, 1985.

36. J.M. Grey. *An Exploration of Musical Timbre*, Ph.D. Dissertation, Stanford University, 1975.

37. J. Strawn. "Approximation and Syntactic Analysis of Amplitude and Frequency Functions for Digital Sound Synthesis," *The Music Machine - Selected Readings from Computer Music Journal*. The MIT Press, pp. 671-692, 1989.

38. B. Mauchly, Software Engineer. *Dinner Conversation at International Computer Music Conference*, October 1991.

39. J. Strawn. "Implementing Table Lookup Oscillators for Music with the Motorola DSP56000 Family," *Preprint for the 85th Convention of the Audio Engineering Society*, 1988.

40. S.D. Peters. *Design of Linear Time-Varying Recursive Digital Filters*, Master's Thesis, Queen's University, 1988.

APPENDIX A
MATLAB SOURCE CODE FOR ANALYSIS PROCEDURE

1. Main Menu

```
% m-file for controlling KL Analysis routines
%
% Call:      kl_main
%
% Method:    menu access to various analysis functions
%
% References:      Stapleton Thesis
%
% Rev. History:    Original Code: May 7, 1991
%                  Modifications: ongoing

% Calls:
% Matlab:    menu, input, clear, clc, clg
%
% KLAnal:    extract_sf, gen_ensemble, extract_synthpars,
%            review_synthpars, re_synth, compare, init_globals

%% Clear, then Initialize global variables
clear, clc, clg
init_globals

%% User interaction main menu
notdone = 1;
while notdone,
   choice = menu('KL Tone Analysis Main Menu', ..
      extract_sfLN,gen_ensembleLN,extract_synthparsLN, ..
      review_synthparsLN,re_synthLN,compareLN,'Quit');
   if choice == 1
      extract_sf
   elseif choice == 2
      gen_ensemble
   elseif choice == 3
      extract_synthpars
   elseif choice == 4
      review_synthpars
   elseif choice == 5
      re_synth
   elseif choice == 6
      compare
   else
      leave = input('Do you really want to quit (y/n)?','s');
      if length(leave) == 1,
         if (leave == 'y') | (leave == 'Y')
            notdone = 0;
```

```
        end
      end
   end
end
```

1.1. Startup Initialization of Global Control Parameters

```
% m-file for initializing KL Analysis global variables
%
% Call:     init_globals
%
% References:  My Notes
%
% Rev. History:   Original Code: May 4, 1991
%                 Modifications: ongoing

% Calls:
% Matlab:   global

%% User-modifiable global variables
%% ================================

% Signal parameters:

global   K_SF      % period of single-period sample functions
global   NUM_SF    % number of sample functions to extract from
                   % tone
global   FS_PB     % playback sampling frequency for synthesis

K_SF     = 256;
NUM_SF   = 8;
FS_PB    = 44.1e3;

% Data Processing parameters:

% number of points at which to estimate period
% for averaged period estimate
global   NUM_PERIOD_ESTS
% number of points from original signal to use
% in re-sampling interpolation step for period
% normalization.
global   NUM_INTERP
% Phase Alignment Threshold
global   PA_THRESH
% size of frequency envelope smoothing window - must be odd
global   SMOOTH_WIN_SIZE
% number of linear segments to use in interpolating
% amplitude and frequency envelopes if using ADJUST
% algorithm.
global   NUM_SEGS
% Maximum iterations in adjust algorithm.
global   ADJUST_MAXITS
% Number of attack segments for env_seg_userin2
global   NUM_ATTACK_SEGS
% Number of decay segments for env_seg_userin2
global   NUM_DECAY_SEGS
```

```
% Number of samples per tick - real-time synth
global   TICK_LENGTH

NUM_PERIOD_ESTS    = 8;
NUM_INTERP         = 8;
PA_THRESH          = .0001;
SMOOTH_WIN_SIZE    = 17;
NUM_SEGS           = 100;
ADJUST_MAXITS      = 2;
NUM_ATTACK_SEGS    = 5;
NUM_DECAY_SEGS     = 4;
% NUM_SEGS         = NUM_ATTACK_SEGS + NUM_DECAY_SEGS + 1;
TICK_LENGTH        = 44;

% Program Control parameters:

% whether or not to pause after on-the-fly graphs
global   PAUSE_ON_GRAPHS
% whether or not code is being run in batch -
% suppresses plots etc. - primarily for running:
%   Extract Synth Pars
%   Re-synthesize
global   BATCH_MODE
% whether or not to use the ADJUST algorithm for
% linearly segmenting envelopes - if not, user
% graphically inputs segments
global   USE_ADJUST
% whether or not to use a flat frequency value, or
% a linearly segmented envelope input by the user.
global   FLAT_FREQUENCY

PAUSE_ON_GRAPHS = 0;
BATCH_MODE      = 1;
USE_ADJUST      = 1;
FLAT_FREQUENCY = 0;

% Data file directory paths:

global   DIR_TONES    % directory of instrument tones
global   DIR_SFS      % directory of instrument sample functions
                      % - sub-directories given by K_SF
global   DIR_ENS      % directory of sample function ensembles
                      % - sub-directories given by K_SF
global   DIR_SYNTHPARS
         % directory of synthesis parameters for ensembles
         % - sub-directories given by K_SF
global   DIR_SYNTHTONES % directory of synthesized tones

DIR_TONES      = '/home/bweeks/InstrumentTones/';
DIR_SFS        = '/home/bweeks/InstrumentSFs/';
DIR_ENS        = '/home/bweeks/Ensembles/';
DIR_SYNTHPARS  = '/home/bweeks/SynthPars/';
DIR_SYNTHTONES = '/home/bweeks/SynthTones/';

%% Internal global variables
%% ========================
```

```
% function long names, for menus/messages

global   extract_sfLN
global   gen_ensembleLN
global   extract_synthparsLN
global   review_synthparsLN
global   savedsp_synthparsLN
global   re_synthLN
global   compareLN

extract_sfLN = 'Extract Sample Functions from an Instrument Tone';
gen_ensembleLN = 'Generate KL Basis for Ensemble of Sample Functions';
extract_synthparsLN = ..
'Extract Synthesis Parameters for Tones in Ensemble';
savedsp_synthparsLN = 'Save Real-Time DSP Synthesis Parameters';
review_synthparsLN = 'Review Synthesis Parameter Extraction';
re_synthLN = 'Re-synthesize a Tone in Non-Realtime';
compareLN = 'Compare Original and Synthesized Tones';
```

2. Extract Sample Functions from an Instrument Tone - Main Routine

```
function extract_sf
% m-func for extracting single-period normalized sample functions from
% the sustain region of an Instrument Tone
%
% Call:     extract_sf
%
% Where:    No Input or Output parameters
%
% Method:   See code.
%
% References:  Stapleton Thesis.
%     My Notes.
%
% Rev. History:   Original Code: May 4, 1991
%     Modified: May 27, 1991
%       - save kp_tone sus_beg sus_end as well as samp_func
%       - extract and re-sample 2-period buffers instead of 4
%     Modified: July 31, 1991
%       - make multiple period estimates across sustain region,
%         number of estimates specified by global NUM_PERIOD_ESTS

% Calls:
% Matlab:   isempty, length, disp, round, save, eval, int2str,
%     fix, median
%
% KLAnal:   load_tone, parse_ntag, ntag_to_pitch, sustain_region,
%     extract_buffers, re_samp, single_period

%% Load user-specified Instrument Tone
[tone, fs_tone, tone_name] = load_tone;
% If user quit load_tone, then quit extract_sf
if isempty(tone)
   disp(['No tone loaded - exitting ' extract_sfLN])
   return
end
```

```
%% Parse Note Tag off of filename
ntag = parse_ntag(tone_name);
% If tone_name does not end with valid Note Tag, quit extract_sf
if length(ntag) == 3
   if ntag == 'BAD',
      disp(['Bad Note Tag at end of ' tone_name ' ...'])
      disp(['Exitting ' extract_sfLN])
      return
   end
end

%% Guess Tone Period from Note Tag
k_guess = round(fs_tone/ntag_to_pitch(ntag));

%% Have User locate Boundaries of Tone Sustain Region
[sus_beg, sus_end] = sustain_region(tone, k_guess);

%% Estimate exact Tone Period from middle of Sustain Region
disp('Initial Estimate of Tone Period from Note Tag in Filename...')
disp(['  Period = ' int2str(k_guess) ' samples'])
disp('Determining Exact Period of Tone at Middle of Sustain Region...')
sus_len = sus_end - sus_beg;
for ests=1:NUM_PERIOD_ESTS
   est_pt = sus_beg + fix((ests-1)/NUM_PERIOD_ESTS * sus_len);
   k_estbuf(ests) = period_est(tone, k_guess, est_pt);
end
k_est = median(k_estbuf);  % use median to remove any wildly
            % off estimates
disp('  Period Estimates:')
disp(k_estbuf)
disp(['  Period = ' int2str(k_est) ' samples'])

%% Extract Short Buffers from Sustain Region
disp('Extracting Short Buffers from Sustain Region of Tone')
buffer = extract_buffers(tone, NUM_SF, 2*k_est, sus_beg, sus_end);

%% Period-normalize Buffers
disp (['Normalize Tone Period to ' int2str(K_SF) ' samples ...'])
for buf = 1:NUM_SF
   disp(['Normalizing Period of Tone Buffer ' int2str(buf)])
   buffer_norm(buf,:) = re_samp(buffer(:,buf), K_SF, k_est, NUM_INTERP);
end
buffer_norm = buffer_norm';

%% Extract Single Period Samples from each Buffer
disp('Extracting Single Period Sample Functions')
for buf = 1:NUM_SF
   samp_func(:,buf) = single_period(buffer_norm(:,buf), K_SF, 1-1e-6);
end

%D subplot(221)
%D for buf = 1:4
%D    plot([samp_func(:,buf) ; samp_func(:,buf)])
%D end

%D [basis, weight] = klt(samp_func);
```

```
%D weight'
%D sum(weight)

%% Save Sample Functions
disp('Saving Single Period Sample Functions in:')
disp(['  ' DIR_SFS int2str(K_SF) '/' tone_name '_sf.mat'])
kp_tone = k_est;
eval(['save ' DIR_SFS int2str(K_SF) '/' tone_name ..
      '_sf samp_func kp_tone sus_beg sus_end'])
```

2.1. Load Tone Subroutine

```
function [tone, fs, tone_name] = load_tone
% m-func for loading a user-specified Instrument Tone .mat file
%
% Call:  [tone, fs, tone_name] = load_tone
%
% Where: tone is the sampled Instrument Tone. Return tone = []
%           if no tone loaded.
%        fs [Hz] is the sample rate for the tone
%        tone_name is the filename of the tone without the
%           .mat extension.
%
%
% Rev. History:   Original Code: May 4, 1991

% Calls:
% Matlab:   clc, home, menu, isempty, input, length, disp, eval, load

clc, home
notdone = 1;
while notdone
   choice = menu('Load Instrument Tone Menu', ..
            'Load Instrument Tone (default)', ..
            'List Instrument Tones, then Load', ..
            'Quit');
   if isempty(choice)
      choice = 1;
   end
   if choice == 1
      tone_name = input('Enter Instrument Tone to Load: ','s');
      % Strip off .mat if included
      if length(tone_name) > 4
         if tone_name(length(tone_name)-3:length(tone_name)) == '.mat'
       tone_name = tone_name(1:length(tone_name)-4);
    end
      end
      if exist([DIR_TONES tone_name '.mat']) == 2
    disp(['Loading ' tone_name '.mat ...'])
         eval(['load ' DIR_TONES tone_name])
    disp('Tone loaded')
         return
      else
    disp(['Can''t find ' DIR_TONES tone_name '.mat'])
      end
   elseif choice == 2
```

```
      disp(' ')
      disp(['Directory Listing of ' DIR_TONES '*.mat'])
      eval(['dir ' DIR_TONES '*.mat'])
      tone_name = input('Enter Instrument Tone to Load: ','s');
      % Strip off .mat if included
      if length(tone_name) > 4
         if tone_name(length(tone_name)-3:length(tone_name)) == '.mat'
       tone_name = tone_name(1:length(tone_name)-4);
    end
      end
      if exist([DIR_TONES tone_name '.mat']) == 2
    disp(['Loading ' tone_name '.mat ...'])
         eval(['load ' DIR_TONES tone_name])
    disp('Tone loaded')
         return
      else
    disp(['Can''t find ' DIR_TONES tone_name '.mat'])
      end
   else     % choice == 3
      tone = [];
      fs = [];
      tone_name = [];
      return
   end
end
```

2.2. Parse Note Tag Subroutine

```
function [ntag] = parse_ntag (string)
% m-func for parsing a ntag string off the end of a string
%
% Call:  ntag = parse_ntag (string)
%
% Where: ntag is:
%        1) the 2 or 3 character parsed string with format:
%           A..G[#]0..9
%        2) 'BAD' indicating invalid ntag at end of string
%           string is a string vector with a ntag at the end of
%           of the string, 'BAD' returned if less than length 2.
%
% Rev. History:   Original Code: Mar 27, 1991

% Calls:
% Matlab:   length

str_len = length(string);
if str_len < 2,
   ntag = 'BAD';
   return
end
if (string(str_len) >= '0') & (string(str_len) <= '9'),
   octave = string(str_len);
else
   ntag = 'BAD';
   return
end
```

```
% look for '#' in second to last character
if string(str_len-1) == '#',
   if (string(str_len-2) >= 'A') & (string(str_len-2) <= 'G'),
      ntag = [string(str_len-2) '#' octave];
      return
   else
      ntag = 'BAD';
      return
   end
else
   if (string(str_len-1) >= 'A') & (string(str_len-1) <= 'G'),
      ntag = [string(str_len-1) octave];
      return
   else
      ntag = 'BAD';
      return
   end
end
```

2.3. Note Tag to Pitch Conversion Subroutine

```
function [pitch] = ntag_to_pitch (ntag)
% m-func for converting string ntag to numerical pitch
%
% Call:  pitch = ntag_to_pitch (ntag)
%
% Where: pitch [Hz] is the pitch represented by ntag in an
%           equal temperament tuning system
%        ntag is a 2 or 3 character string with format:
%           A..G[#]0..9
%           N.B. no checking is performed on this string
%
% References:  "Musical Application of Microprocessors", pp. 14-15
%                 Hal Chamberlin
%              "Elements of Computer Music", p. 485
%                 F. Richard Moore
%
% Rev. History:   Original Code: Mar 26, 1991

% Calls:
% Matlab:   abs, length

freqC0 = 2^0.25 * 27.5/2;
basenum = [9 11 0 2 4 5 7];   % define notenums for A..G, relative to C
ib = abs(ntag(1)) - abs('A') + 1;
if length(ntag) == 2,
   notenum = basenum(ib);
   octavenum = abs(ntag(2)) - abs('0');
else, % assume length(ntag) == 3 and ntag(2) = '#'
   notenum = basenum(ib) + 1;
   octavenum = abs(ntag(3)) - abs('0');
end
pitch = 2^((octavenum*12 + notenum)/12) * freqC0;
```

2.4. User Selects Sustain Region Subroutine

```
function [sus_beg, sus_end, env] = sustain_region(tone, k_guess)
% m-func for determining sustain region of tone
%
% Call:  [sus_beg, sus_end, env] = sustain_region(tone, k_guess)
%
% Where: sus_beg is the sample value at the beginning of the
%           tone's sustain region.
%        sus_end is the sample value at the end of the
%           tone's sustain region.
%        env is the max amp env of the tone
%        tone is the tone.
%        k_guess is a reasonable guess at the tone's period.
%
% Method:   plot amplitude envelope of tone, and get user to point
%           at beginning and end of tone.
%
% References:  "On the Automatic Transcription of Percussive Music ..."
%              by W. Andrew Schloss, pp. 58-59.
%
% Rev. History:   Original Code: May 4, 1991
%     Modified: December 6, 1991
%     - returns env for documentation (thesis) uses!

% Calls:
% Matlab:   fix, length, clg, plot, title, disp, ginput, sort

disp('Identify Tone Sustain Region ...')

%% Calculate maximum amplitude envelope
seg_len = 2*k_guess;
k_env = 0;
for k = 1:seg_len:seg_len*fix(length(tone)/seg_len),
   k_env = k_env+1;
   env(k_env) = max(abs(tone(k:k+seg_len-1)));
end

%% Plot envelope
clg
plot(env), title('Amplitude Envelope of Instrument Tone')

%% Query user for endpoints of Sustain Region
disp('Use mouse to click at beginning and end of Tone Sustain Region')
[x,y] = ginput(2);
x = sort(x);
sus_beg = fix(seg_len*x(1));
sus_end = fix(seg_len*x(2));
disp('Tone Sustain Region selected')
```

2.5. Period Estimation Subroutine

```
function [k_est] = period_est (s, k_guess, k_start)
% m-func for estimating the period of a near-periodic tone
```

```
%
% Call:  k_est = period_est (s, k_guess, k_start)
%
% Where: k_est [Hz] is the estimated period of the tone computed.
%        s is the tone.
%        k_guess [Hz] provides the center period for the search.
%        k_start [sample] defines the point in the tone sample from
%           which to commence the search, typically in the
%           sustain region of the tone.
%
% Method:   Correlation method.
%
% References:  Stapleton Thesis
%
% Rev. History:   Original Code: January, 1991

% Calls:
% Matlab:   round, size, max

%% Restrict search range to between kmin and kmax
kmin = round(0.5*k_guess);
kmax = round(1.5*k_guess);

%% Find first zero-crossing after start point
k = k_start;
while s(k) == 0,      % move to positive or negative region
   k = k+1;
end
if s(k) > 0,
   while s(k) >= 0,
      k = k+1;
   end
   slope = '-';
else
   while s(k) <= 0,
      k = k+1;
   end
   slope = '+';
end
k0 = k;

%% Extract search buffer - twice the length of the maximum period -
starting
%% at the first zero-crossing after the start point
buffer_length = 2 * kmax;
sb = s(k0:k0+buffer_length-1);

%% Find all zero-crossings with same slope in search buffer
k = kmin;
k_last = kmax + 1;
num_zc = 0;
% Positive slope crossings:
if slope == '+',
   while sb(k) >= 0,     % Initially get to negative portion
      k = k+1;
   end
   while k <= k_last,
```

```
      % scan negative portion of signal till positive-going zero-
      % crossing found, or end of search
      while (sb(k) <= 0) & (k <= k_last),
         k = k+1;
      end
      if k > k_last,
         break
      % record location of zero-crossing
      else
         num_zc = num_zc+1;
         zc(num_zc) = k;
         k = k+1;
      end
      % scan positive portion of signal till negative-going zero-
      % crossing found, or end of search
      while (sb(k) >= 0) & (k <= k_last),
         k = k+1;
      end
      if k > k_last,
         break
      else
         k = k+1;
      end
      % loop to find more positive-going zero-crossings
   end
% Negative slope crossings:
else  % slope = '-'
   while sb(k) <= 0,     % Initially get to positive portion
      k = k+1;
   end
   while k <= k_last,
      % scan positive portion of signal till negative-going zero-
      % crossing found, or end of search
      while (sb(k) >= 0) & (k <= k_last),
         k = k+1;
      end
      if k > k_last,
         break
      % record location of zero-crossing
      else
         num_zc = num_zc+1;
         zc(num_zc) = k;
         k = k+1;
      end
      % scan negative portion of signal till positive-going zero-
      % crossing found, or end of search
      while (sb(k) <= 0) & (k <= k_last),
         k = k+1;
      end
      if k > k_last,
         break
      else
         k = k+1;
      end
      % loop to find more negative-going zero-crossings
   end
end
```

```
%D Debugging statements
% num_zc
% slope
% zero_crossings = zc + k0-1

%% Compute correlations for each zero-crossing found
[m,n] = size(sb);
if m==1,        % sb is a row vector
   sb = sb';
end
% sb is a column vector now for sure
for p=1:num_zc,
   c(p) = sb(1:k_guess)'*sb(zc(p):zc(p)+k_guess-1);
end
%% Choose period exhibiting highest correlation
[junk, p] = max(c);
k_est = zc(p) - 1;

%% End
```

2.6. Buffer Extraction Subroutine

```
function buffer = ..
extract_buffers(tone, num_buf, buf_len, reg_beg, reg_end)
% m-func for extracting short buffers from region of tone
%
% Call:  buffer = extract_buffers(tone,num_buf,buf_len,reg_beg,reg_end)
%
% Where: buffer is buf_len x num_buf matrix, each column being
%           a sample buffer from the tone.
%        tone is the samples from which to extract buffers.
%        num_buf is the number of buffers to extract.
%        buf_len is the length of buffers to extract.
%        reg_beg and reg_end mark the region of tone from which to
%           extract short buffers.
%
% Method:   See code.
%
% References:  Stapleton Thesis.
%
% Rev. History:   Original Code: May 6, 1991

% Calls:
% Matlab:   size, fix

%% If tone is a row vector, convert to column vector
[rows, cols] = size(s);
if rows == 1,
   s = s';
end

%% Extract evenly spaced buffers in region
buffer = zeros(buf_len, num_buf);
buf_sep = fix((reg_end - reg_beg)/num_buf);
for buf = 1:num_buf
   k = reg_beg + (buf-1)*buf_sep;
```

```
   buffer(:,buf) = tone(k:k+buf_len-1);
end
```

2.7. Sample Rate Conversion Subroutine

```
function [sr, h] = re_samp (s, L, M, Q)
% m-func for re_sampling signal, scaling original sample rate by
%  scale_factor = L/M
%
% Call:  sr = re_samp (s, L, M, Q)
%        [sr, h] = re_samp (s, L, M, Q)
%
% Where: sr is re-sampled signal - row vector
%        h is the interpolating filter impulse response
%           (optional return)
%
%     s is the original signal - row or column vector
%     L is the sample rate increase to perform
%     M is the sample rate decrease to perform
%     Q is the number of samples to include in each interpolation
%
% Method:   Re-sample successive length M buffers of s into
%     length L buffers of sr.  Uses polyphase FIR structure
%     for re-sampling filter, efficiently combining the
%     interpolation and decimation procedures.  Bandlimits
%     signal s if scale_factor < 1.  FIR is length Q * L.
%     FIR designed using fir1 procedure and Blackman window.
%     Bandwidth of filter is min(1/L, 1/M).
%
% References:  "Interpolation and Decimation of Digital Signals",
%              R.E. Crochiere and L.R. Rabiner,
%              Proceedings of the IEEE, Vol. 69, No. 3, March 1981.
%
% Rev. History:   Original code - February 21, 1991.

% Calls:
% Matlab:   min, zeros, rem, length, fix, fliplr
%
% Signal:   fir1, blackman

%% If s is column vector, convert to row vector
[rows, cols] = size(s);
if cols == 1,
   s = s';
end

%% design FIR filter h - extract polyphase filters p
h = fir1(Q*L - 1, min([1/L 1/M]), blackman(Q*L));
p = zeros(L,Q);
for l = 1:L,
   Mp = rem((l-1)*M, L) + 1;
   for q = 1:Q,
      p(l,q) = h(Mp + (q-1)*L);
   end
end
```

```
%% append Q-1 zeros to signal s
s(length(s)+1:length(s)+Q-1) = zeros(1,Q-1);

%% re-sample s, block by block
num_full_blks = fix(length(s)/M);    % number of complete length M blocks
s_win = zeros(1,Q);                  % init. interpolation window
sr_blk = zeros(1,L);                 % init. temporary output block
sr = zeros(1,fix(length(s)*L/M));    % init. entire output buffer

%% re-sample first num_full_blks-1 blocks
%D num_full_blks
for blk = 1:num_full_blks-1,
   s_blk = s((blk-1)*M + 1:blk*M);
   shifted = 0;
   for l = 1:L,
      shiftval = fix ((l-1)*M/L) + 1 - shifted;
      if shiftval > 0,
         if shiftval >= Q, % no overlap between prev. and curr. window
            s_win =
               fliplr(s_blk(shifted+shiftval-Q+1:shifted+shiftval));
         else              % overlap between prev. and curr. window
            s_win(shiftval+1:Q) = s_win(1:Q-shiftval);
            s_win(1:shiftval) =
               fliplr(s_blk(shifted+1:shifted+shiftval));
         end
         shifted = shifted + shiftval;
      end
      sr_blk(l) = s_win * p(l,:)';
   end
   s_win = fliplr(s_blk(M-Q+1:M));
   sr((blk-1)*L + 1:blk*L) = L*sr_blk;    % required filter gain
end

%% re-sample last full block plus extra partial block
blk = num_full_blks;
s_blk = s((blk-1)*M + 1:length(s));
out_blk_size = length(sr) - (blk-1)*L;
sr_blk = zeros(1,out_blk_size);     % init. new temporary output block
shifted = 0;
for l = 1:out_blk_size,
   shiftval = fix ((l-1)*M/L) + 1 - shifted;
   if shiftval > 0,
      if shiftval >= Q,     % no overlap between prev. and curr. window
         s_win = fliplr(s_blk(shifted+shiftval-Q+1:shifted+shiftval));
      else         % overlap between prev. and curr. window
         s_win(shiftval+1:Q) = s_win(1:Q-shiftval);
         s_win(1:shiftval) = fliplr(s_blk(shifted+1:shifted+shiftval));
      end
      shifted = shifted + shiftval;
   end
   fil_index = rem(l,L);
   if fil_index == 0,
      fil_index = L;
   end
   sr_blk(l) = s_win * p(fil_index,:)';
end
sr((blk-1)*L + 1:length(sr)) = L*sr_blk;  % required filter gain
```

2.8. Derive Single Period Sample Functions Subroutine

```
function [sp_samp] = single_period (buffer, k_sp, fc)
% m-func for extracting single period, unit-energy normalized sample
% function from a buffer.
%
% Call:  sp_samp = single_period (buffer, k_sp, fc)
%
% Where: sp_samp is a column vector containing the single period
%           sample extracted from buffer.
%        buffer is a multi-period buffer from which to extract
%           sp_samp.
%        k_sp is the length of a single period.
%        fc is the cutoff frequency of the smoothing low-pass,
%           values ranging from 0<fc<1.
%
% Method:   See code.
%
% References:  Stapleton Thesis.
%
% Rev. History:   Original Code: May 6, 1991

% Calls:
% Matlab:   length, norm
%
% Signal:   fir1, filter

%% Move into buffer half a period and extract single period
s = buffer(fix(k_sp/2):length(buffer));
s = s(1:k_sp);
% Make s a row vector, if necessary
[rows, cols] = size(s);
if cols == 1,
   s = s';
end

%% Smooth s and discard first period
sf = filter(fir1(30,fc),1,[s s s]);
sf = sf(k_sp:length(sf));

%% Find first zero-crossing after beginning of sf buffer and extract
%% single-period following zero-crossing.
k = 1;
while sf(k) == 0,     % move to positive or negative region
   k = k+1;
end
if sf(k) > 0,
   while sf(k) >= 0,
      k = k+1;
   end
else
   while sf(k) <= 0,
      k = k+1;
   end
end
sf = sf(k:k+k_sp-1);
```

```
%% Normalize energy of single period sample
c = 1 / norm(sf);
sp_samp = c * sf';
```

3. Generate KL Basis for Ensemble of Sample Functions - Main Routine

```
function gen_ensemble
% m-func for concatenating single-period normalized sample functions
% into an ensemble of phase-aligned sample functions from several
% Instrument Tones, then computing KL Basis Functions.
%
% Call:  gen_ensemble
%
% Where: No Input or Output parameters
%
% Method:   See code.
%
% References:  Stapleton Thesis.
%              My Notes.
%
% Rev. History:   Original Code: May 7, 1991
%     Modified: June 4, 1991
%     - modified plots echoing to user
%     Modified: March 6, 1992
%     - passed unaligned ensemble to save_ens for documentation
%       purposes
%     Modified: April 23, 1992
%     - compute retained energy per additional basis functions
%       for passing to save_ens for documentation purposes

% Calls:
% Matlab:   disp, plot, title, sum, num2str
%
% KLAnal:   load_sfs, phase_align, klt, save_ens, mypause

%% Load unaligned ensemble of user-specified Instrument Sample Functions
[ensemble_ua, tone_names, num_sfs] = load_sfs;
% If user quit load_sf, then quit gen_ensemble
if isempty(ensemble_ua)
   disp(['No sample functions loaded - exitting ' gen_ensembleLN])
   return
end
%D tone_names
%D num_sfs
%D plot(ensemble_ua),title('Unaligned ensemble')
%D mypause

%% Phase-align ensemble
disp('Phase Aligning Ensemble ...')
[ensemble, num_its] = phase_align(ensemble_ua);
clg
subplot(211),plot(ensemble),title('Phase Aligned Ensemble')

%% Compute basis for ensemble
```

```
disp('Computing KL Basis Functions for Ensemble ...')
[basis, weight] = klt(ensemble);
for k=1:length(weight)
   weighted_basis(:,k) = sqrt(weight(k)) * basis(:,k);
end
% Show results
subplot(212),plot(weighted_basis), ..
  title('Ensemble KL Basis Functions (Weighted)')
retained=zeros(length(weight),1);
for k=1:length(weight)
   retained(k) = sum(weight(1:k));
end
disp('First 8 KL Basis Function Weights and Retained Energy')
if length(weight) > 12
   disp([weight(1:12) retained(1:12)])
else
   disp([weight retained])
end
disp(['Total Ensemble Energy = ' num2str(sum(weight))])

%% Save phase-aligned ensemble and basis functions
save_ens(ensemble, ensemble_ua, tone_names, num_sfs, basis, weight, ..
retained)
```

3.1. Load Single Period Sample Functions Subroutine

```
function [ensemble, tone_names, num_samps] = load_sfs
% m-func for loading multiple user-specified Sample Function .mat files
%
% Call:  [ensemble, tone_names] = load_sfs
%
% Where: ensemble is an ensemble of sample functions loaded from
%           1 or more Sample Function files - each column is a
%           single period sample function.  If user quits load_sfs
%           before loading any sample functions, then ensemble = [].
%        tone_names is a matrix of Tone Names, 1 Tone Name per
%           row (left-justified, trailing blanks), each Tone Name
%           corresponding to a Sample Function .mat file included
%           in ensemble.
%        num_samps is a row vector, the ith element indicating
%           how many samples in the ensemble are from the ith
%           tone_name.
%
% Rev. History:   Original Code: May 7, 1991

% Calls: clc, home, menu, isempty
% Matlab:

clc, home
notdone = 1;
loaded = 0;
ensemble = [];
tone_names = [];
num_samps = [];
while notdone
   choice = menu('Load Sample Functions Menu', ..
```

```
        'Load Sample Function (default)', ..
        'List Sample Functions, then Load', ..
        'Quit Loading');
if isempty(choice)    % set default case
   choice = 1;
end
if choice == 1
   tone_name = input('Enter Sample Functions File to Load: ','s');
   % Strip off .mat if included
   if length(tone_name) > 4
      if tone_name(length(tone_name)-3:length(tone_name)) == '.mat'
    tone_name = tone_name(1:length(tone_name)-4);
 end
   end
   % Strip off _sf if included
   if length(tone_name) > 3
      if tone_name(length(tone_name)-2:length(tone_name)) == '_sf'
         tone_name = tone_name(1:length(tone_name)-3);
      end
   end
   if exist([DIR_SFS int2str(K_SF) '/' tone_name '_sf.mat']) == 2
      disp(['Loading ' tone_name '_sf.mat ...'])
      eval(['load ' DIR_SFS int2str(K_SF) '/' tone_name '_sf'])
      loaded = 1;
   else
      disp(['Can''t find ' DIR_SFS int2str(K_SF) '/' ..
            tone_name '_sf.mat'])
   end
elseif choice == 2
   disp(' ')
   disp(['Directory Listing of ' DIR_SFS int2str(K_SF) '/*.mat'])
   eval(['dir ' DIR_SFS int2str(K_SF) '/*.mat'])
   tone_name = input('Enter Instrument Tone to Load: ','s');
   % Strip off .mat if included
   if length(tone_name) > 4
      if tone_name(length(tone_name)-3:length(tone_name)) == '.mat'
         tone_name = tone_name(1:length(tone_name)-4);
      end
   end
   % Strip off _sf if included
   if length(tone_name) > 3
      if tone_name(length(tone_name)-2:length(tone_name)) == '_sf'
         tone_name = tone_name(1:length(tone_name)-3);
      end
   end
   if exist([DIR_SFS int2str(K_SF) '/' tone_name '_sf.mat']) == 2
      disp(['Loading ' tone_name '_sf.mat ...'])
      eval(['load ' DIR_SFS int2str(K_SF) '/' tone_name '_sf'])
      loaded = 1;
   else
      disp(['Can''t find ' DIR_SFS int2str(K_SF) '/' ..
            tone_name '_sf.mat'])
   end
else     % choice == 3
   return
end
% if a sample function was loaded
```

```
   if loaded
      if isempty(ensemble)
         ensemble = samp_func;
         tone_names = tone_name;
         [rowsf,colsf] = size(samp_func);
         num_samps(1) = colsf;
      else
         % append sample functions to ensemble
         [rowsf,colsf] = size(samp_func);
         [rowens,colens] = size(ensemble);
         % assume same sample length (number of rows) for ensemble
         % and samp_func, if not we have a big problem!!!
         ensemble(:,colens+1:colens+colsf) = samp_func;
         % append tone_name to tone_names
         [num_names, length_names] = size(tone_names);
         if length(tone_name) == length_names
            tone_names(num_names+1,:) = tone_name;
         elseif length(tone_name) < length_names
            % append spaces to tone_name
            for ci = length(tone_name)+1:length_names
               tone_name(ci) = ' ';
            end
            tone_names(num_names+1,:) = tone_name;
         else
            % append spaces to all names in tone_names
            for ni = 1:num_names
               for ci = length_names+1:length(tone_name)
                  tone_names(ni,ci) = ' ';
               end
            end
            tone_names(num_names+1,:) = tone_name;
         end
         % note how many samples from latest tone
         num_samps(num_names+1) = colsf;
      end
      loaded = 0;
      clear tone_name samp_func
   end
   disp('Ensemble now contains:')
   for sf = 1:length(num_samps)
      disp([int2str(num_samps(sf)) ' sample functions from ' ..
            tone_names(sf,:)])
   end
end
```

3.2. Phase Alignment Subroutine

```
function [ens_pa, num_iterations] = phase_align (ens)
% m-func for phase-aligning an ensemble of single-period sample
% functions
%
% Call:  ens_pa = phase_align (ens)
%
% Where: ens_pa is the phase-aligned ensemble returned.
%        num_iterations is the number of iterations required
%           to perform phase-alignment.
```

```
%        ens is the ensemble of single-period sample functions
%           to phase-align.
%
% Method:    See code.
%
% References:  Stapleton Thesis
%
% Rev. History:    Original Code: May 9, 1991

% Calls:
% Matlab:    svd, max

notdone = 1;
eigval1 = 0;
num_iterations = 0;
max_iterations = 30;
[k_sf, num_sfs] = size(ens);
corrs = zeros(k_sf,num_sfs);
while notdone
   [U,S,V] = svd(ens,0);
   if (S(1,1)>(eigval1 + PA_THRESH)) & (num_iterations < max_iterations)
      num_iterations = num_iterations + 1;
      eigval1 = S(1,1);
      eigvec1 = U(:,1);
      % double period of ens sample functions
      ens = [ens;ens];
      for k = 1:k_sf
         % compute correlations for all sample functions at phase k
         corrs(k,:) = eigvec1' * ens(k:k+k_sf-1,:);
      end
      % locate maximum correlation and corresponding phase for each
      % sample function
      [max_corr, max_loc] = max(corrs);
      for sf = 1:num_sfs,
         ens(1:k_sf,sf) = ens(max_loc(sf):max_loc(sf)+k_sf-1,sf);
      end
      % reduce ens back to single period sample functions
      ens = ens(1:k_sf,:);
   elseif num_iterations >= max_iterations
      disp([int2str(max_iterations) ' iterations reached:'])
      disp('  No further Phase Alignment will be performed.')
      notdone = 0;
   else
      disp(['Phase Alignment converged in ' ..
             int2str(num_iterations) ' iterations.'])
      notdone = 0;
   end
end
ens_pa = ens;
return
```

3.3. KLT Subroutine

```
function [basis, weight] = klt (sample_funcs)
% m-func for determining the KL Transform of sample functions.
%
```

```
% Call:  [basis, weight] = klt (sample_funcs)
%
% Where: basis is the MxN matrix of N basis functions,
%           each column being a basis function.
%        weight is a length N vector, the kth element being
%           the KLT coefficient of the the kth basis function.
%           That is, the kth element is the eigenvalue corresponding
%           to the kth eigenvector of the MxM covariance matrix
%           for the sample functions.  Note that the weights are
%           returned in order from largest to smallest.
%        sample_funcs is MxN matrix of N sample functions,
%           each column vector being a sample function.
%
% Method:   The singular value decomposition (svd) is applied to
%           sample_funcs.  The economy form of svd is used.
%
% References:  "Matrix Computations" by David Watkins, Chapter 6.
%              "Modern Filters" by Simon Haykin.
%              Stapleton Thesis.
%
% Rev. History:   Original Code: May 2, 1991

% Calls:
% Matlab:   svd, diag, length

[basis,S,V] = svd(sample_funcs,0);
weight = diag(S);
numbasis = length(weight);
for k = 1:numbasis
   weight(k) = (1/numbasis) * (weight(k)^2);
end
```

3.4. Save Ensemble Subroutine

```
function save_ens ..
(ensemble,ens_ua,tone_names,num_sfs,basis,weight,retained)
% m-func for saving Sample Functions Ensemble and KL Basis
%
% Call:  save_ens (ensemble,ens_ua,tone_names,num_sfs,basis,weight, ..
%          retained)
%
% Where: ensemble is an ensemble of sample functions loaded from
%           1 or more Sample Function files and phase aligned -
%           each column is a single period sample function.
%        ens_ua is the ensemble prior to alignment.
%        tone_names is a matrix of Tone Names, 1 Tone Name per
%           row (left-justified, trailing blanks), each Tone Name
%           corresponding to a Sample Function .mat file included
%           in ensemble.
%        num_sfs is a row vector, the ith element indicating
%           how many samples in the ensemble are from the ith
%           tone_name.
%        basis is the MxN matrix of N basis functions,
%           each column being a basis function.
%        weight is a length N vector, the kth element being
%           the KLT coefficient of the the kth basis function.
```

```
%           That is, the kth element is the eigenvalue corresponding
%           to the kth eigenvector of the MxM covariance matrix
%           for the sample functions.
%        retained is vector of successively increasing retained energy
%           for adding one more basis function.
%
% Method:   See Code
%
% Rev. History:   Original Code: May 13, 1991
%     Modified: March 6, 1992
%     - added ens_ua as a parameter for purposes of
%       documenting phase-alignment procedure
%     Modified: April 23, 1992
%     - added retained as a parameter for purposes of documenting
%       retained energy per additional basis function

% Calls:
% Matlab:   disp, eval, input, int2str, isempty, length, exist, save

disp('Current Ensembles Directory Listing:')
eval(['dir ' DIR_ENS int2str(K_SF)])
notdone = 1;
while notdone
   filename = input ..
   ('Input name of file to save Ensemble in (<CR> to not save): ','s');
   if isempty(filename)
      dontsave = input ..
      ('You really don''t want to save the Ensemble (y/n)? ','s');
      if length(dontsave) == 1
         if (dontsave == 'y') | (dontsave == 'Y')
            disp('Ensemble not saved.')
            return
         end
      end
   else
      % Strip off .mat if included
      if length(filename) > 4
         if filename(length(filename)-3:length(filename)) == '.mat'
            filename = filename(1:length(filename)-4);
         end
      end
      % Check if filename already exists and confirm overwrite
      if exist([DIR_ENS int2str(K_SF) '/' filename '.mat']) == 2
         overwrite = input ..
         (['Overwrite existing file: ' filename '.mat (y/n)? '],'s');
         if length(overwrite) == 1
            if (overwrite == 'y') | (overwrite == 'Y')
               notdone = 0;
            end
         end
      else
         notdone = 0;
      end
   end
end
disp(['Saving Ensemble in ' DIR_ENS int2str(K_SF) '/' filename '.mat'])
eval (['save ' DIR_ENS int2str(K_SF) '/' filename ..
```

```
'.mat ensemble ens_ua tone_names num_sfs basis weight retained'])
```

4. Extract Synthesis Parameters for Tones in Ensemble - Main Routine

```
function extract_synthpars
% m-func for extracting the synthesis parameters for all the instrument
% tones in an ensemble
%
% Call:  extract_synthpars
%
% Where: No Input or Output parameters
%
% Method:   See code
%
% References:  Stapleton Thesis.
%              My Notes.
%
% Rev. History:   Original Code: May 30, 1991
%     Modified: August 8, 1991
%     - Save intermediate analysis results for reviewing
%     Modified: August 13, 1991
%     - fs_init assigned first value of f returned from
%       freq_synthpars, instead of first value of f_env
%     Modified: August 24, 1991
%     - intermediate analysis results saved with final results
%       instead of separate file.
%     Modified: September 25, 1991
%     - do not plot if global BATCH_MODE ~= 0
%     - do not clear sus_end
%     Modified: April 23, 1992
%     - label frequency env plot as "smoothed", reflecting
%       change in extract_envs.m
%     Modified: April 30, 1992
%     - fixed mkdir to handle # as part of directory name - fell
%       apart when running in batch mode!!!

% Calls:
% Matlab:   isempty, disp, find, exist, eval, load, clear
%
% KLAnal:   load_ensemble, extract_envs, amp_synthpars, freq_synthpars

%% Load Ensemble for which to generate synthesis parameters
[ens_name, tone_names, basis, weight] = load_ensemble;
% If user quit load_ensemble, then quit extract_synthpars
if isempty(ens_name)
   disp(['No ensemble loaded - exitting ' extract_synthparsLN])
   return
end

%% Query user for how many basis functions to use
disp('Basis function weights:')
disp(weight')
num_basis = ..
input('Input number of basis functions to use for synthesis: ');
```

```
%% Make Synthesis Parameter directory for ensemble
disp(['Making Synthesis Parameter Directory for ' ens_name])
dir_name = ens_name;
sharp = find(dir_name == '#');
if (length(sharp) == 1)
   dir_name = [dir_name(1:sharp-1) '\'
dir_name(sharp:length(dir_name))];
end
eval(['!rm -r ' DIR_SYNTHPARS int2str(K_SF) '/' dir_name])
eval(['!mkdir ' DIR_SYNTHPARS int2str(K_SF) '/' dir_name])

%% Generate Synthesis Parameters for each Instrument Tone represented
%% in Ensemble
for tone_name = tone_names'
   % parse off any trailing blanks on left-justified tone_name
   blanks = find (tone_name == ' ');
   if isempty(blanks)
      tone_name = tone_name';
   else
      tone_name = tone_name(1:blanks(1)-1)';
   end
   disp(['Extracting Synthesis Parameters for ' tone_name ' ...'])
   %% Load Instrument Tone file - creates variables: tone, fs
   if exist([DIR_TONES tone_name '.mat']) == 2
      disp(['Loading ' tone_name '.mat ...'])
      eval(['load ' DIR_TONES tone_name])
   else
      disp(['Can''t find ' DIR_TONES tone_name '.mat'])
      disp(['Exitting ' extract_synthparsLN])
      return
   end

   %% Load Instrument Sample Functions file - create variables:
   %% samp_func, kp_tone, sus_beg, sus_end
   if exist([DIR_SFS int2str(K_SF) '/' tone_name '_sf.mat']) == 2
      disp(['Loading ' tone_name '_sf.mat ...'])
      eval(['load ' DIR_SFS int2str(K_SF) '/' tone_name '_sf'])
   else
      disp(['Can''t find ' DIR_SFS int2str(K_SF) '/' ..
            tone_name '_sf.mat'])
      disp(['Exitting ' extract_synthparsLN])
      return
   end
   % clear unused variables which were loaded
   clear samp_func

   %% Extract Envelopes for tone and echo to user
   [amp_envs, kphase, kphase_uw, f_env, bp] = ..
      extract_envs (basis(:,1:num_basis), tone, sus_beg, kp_tone);
   if ~BATCH_MODE
      clg
      subplot(221),plot(bp,kphase),title('Time-varying Phase (Wrapped)')
      subplot(222),plot(bp,amp_envs), ..
        title('Basis Function Amplitude Envelopes')
      subplot(223),plot(bp,kphase_uw),title('Unwrapped Phase')
      disp(['Estimated frequency of tone = ' num2str(fs/kp_tone) ' Hz'])
      subplot(224),plot(bp,f_env),title('Smoothed Frequency Envelope')
```

```
      if PAUSE_ON_GRAPHS
         mypause
      end
   end

   %% Compute Amplitude Synthesis Parameters
   disp('Computing Amplitude Synthesis Parameters ...')
   [dys, dysbp, y] = amp_synthpars(amp_envs,bp,sus_beg,sus_end,fs);
   y_init = amp_envs(1,:);

   %% Compute Frequency Synthesis Parameters
   disp('Computing Frequency Synthesis Parameters ...')
   [dfs, dfsbp, f] = freq_synthpars(f_env,bp,sus_beg,fs);
   kphase_init = round(K_SF*kphase(1)/kp_tone);
   fs_init = f(1) * K_SF / FS_PB;
   fs_pb = FS_PB;

   %% Save synthesis parameters if file specified by tone_name
   disp('Saving Synthesis Parameters in:')
   disp(['  ' DIR_SYNTHPARS int2str(K_SF) '/' ens_name '/' ..
         tone_name '.mat'])
   eval(['save ' DIR_SYNTHPARS int2str(K_SF) '/' ens_name '/' ..
      tone_name ..
      ' y_init dys dysbp kphase_init fs_init dfs dfsbp basis fs_pb' ..
      ' bp amp_envs f_env kphase kphase_uw fs kp_tone y f'])
end
```

4.1. Load Ensemble Subroutine

```
function [ens_name,tone_names,basis,weight,num_sfs,ensemble] = ..
load_ensemble
% m-func for loading a user-specified Ensemble .mat file
%
% Call:     [ens_name,tone_names,basis,weight,num_sfs,ensemble] ..
%           = load_ensemble
%
% Where: ens_name is the filename of the ensemble loaded (without
%           .mat extension).  If user quits load_ensemble before
%           loading any sample functions, then ens_name = [].
%        tone_names is a matrix of Tone Names, 1 Tone Name per
%           row (left-justified, trailing blanks), each Tone Name
%           corresponding to a Sample Function .mat file included
%           in ensemble.
%        basis is the MxN matrix of N basis functions,
%           each column being a basis function.
%        weight is a length N vector, the kth element being
%           the KLT coefficient of the the kth basis function.
%           That is, the kth element is the eigenvalue corresponding
%           to the kth eigenvector of the MxM covariance matrix
%           for the sample functions.
%        num_sfs is a row vector, the ith element indicating
%           how many samples in the ensemble are from the ith
%           tone_name.
%        ensemble is an ensemble of sample functions loaded from
%           1 or more Sample Function files - each column is a
%           single period sample function.
```

```
%
% Rev. History:   Original Code: May 30, 1991
%     Revised: June 10, 1991, to return ens_name

% Calls:
% Matlab:   clc, home, menu, isempty, input, length, disp, eval, load

clc, home
notdone = 1;
while notdone
   choice = menu('Load Ensemble Menu', ..
            'Load Ensemble (default)', ..
            'List Ensembles, then Load', ..
            'Quit');
   if isempty(choice)
      choice = 1;
   end
   if choice == 1
      ens_name = input('Enter Ensemble to Load: ','s');
      % Strip off .mat if included
      if length(ens_name) > 4
         if ens_name(length(ens_name)-3:length(ens_name)) == '.mat'
            ens_name = ens_name(1:length(ens_name)-4);
         end
      end
      if exist([DIR_ENS int2str(K_SF) '/' ens_name '.mat']) == 2
         disp(['Loading ' ens_name '.mat ...'])
         eval(['load ' DIR_ENS int2str(K_SF) '/' ens_name])
         disp('Ensemble loaded')
         return
      else
         disp(['Can''t find ' DIR_ENS int2str(K_SF) '/' ..
            ens_name '.mat'])
      end
   elseif choice == 2
      disp(' ')
      disp(['Directory Listing of ' DIR_ENS int2str(K_SF) '/' '*.mat'])
      eval(['dir ' DIR_ENS int2str(K_SF) '/' '*.mat'])
      ens_name = input('Enter Ensemble to Load: ','s');
      % Strip off .mat if included
      if length(ens_name) > 4
         if ens_name(length(ens_name)-3:length(ens_name)) == '.mat'
            ens_name = ens_name(1:length(ens_name)-4);
         end
      end
      if exist([DIR_ENS int2str(K_SF) '/' ens_name '.mat']) == 2
         disp(['Loading ' ens_name '.mat ...'])
         eval(['load ' DIR_ENS int2str(K_SF) '/' ens_name])
         disp('Ensemble loaded')
         return
      else
         disp(['Can''t find ' DIR_ENS int2str(K_SF) '/' ..
            ens_name '.mat'])
      end
   else     % choice == 3
      ens_name = [];
      tone_names = [];
```

```
      basis = [];
      weight = [];
      num_sfs = [];
      ensemble = [];
      return
   end
end
```

4.2. Extract Amplitude and Frequency Envelopes Subroutine

```
function [amp_envs, kphase, kphase_uw, f_env, bp] = ..
   extract_envs (basis, tone, sus_beg, kp_tone)
% m-func for extracting the amplitude envelopes and time-varying phase
% and frequency for a tone.
%
% Call:     [amp_envs, kphase, kphase_uw, f_env, bp] = ..
%              extract_envs (basis, tone, sus_beg, kp_tone)
%
% Where: amp_envs is a MxN matrix, the nth column containing the
%           amplitude envelope of the nth basis function.
%        kphase is a length M vector containing the time-varying
%           phase for all of the basis functions - phase values
%           are integers ranging from 1 through kp_tone.
%        kphase_uw is a length M vector containing the unwrapped
%           phase for all of the basis functions.
%        f_env is a length M vector containing the frequency
%           envelope computed from kphase_uw and then smoothed to
%           remove effects of quantized phase values.
%        bp is a length M vector containing the breakpoint sample
%           values for amp_envs and kphase.
%        basis is the (truncated) KL basis, 1 basis function per
%           column.
%        tone is the sampled instrument tone for which to extract
%           the amplitude envelope and time-varying phase
%        sus_beg is the estimated beginning of the tone's sustain
%           region.
%        kp_tone is the estimated average period of the tone.
%
% Method:   See code.
%
% References:  Stapleton Thesis.
%              My Notes.
%
% Rev. History:   Original Code: May 31-June 10, 1991
%     Modified: August 26, 1991
%     - sets first frequency envelope value to median of
%       all values, to avoid wild point
%     Modified: September 24, 1991
%     - intermittent bug fix - dimension of basis1vpm
%     Modified: April 23, 1992
%     - smooth frequency envelope using a Hamming window
%       to provide weighted average of surrounding frequency
%       points to compute each separate point.

% Calls:
% Matlab:   spline, max, round, fliplr, flipud, norm
```

```
%% Re-sample basis functions to length kp_tone using spline
%D disp(['Period of Tone = ' int2str(kp_tone)])
disp(['Re-sampling basis functions to tone period length ' ..
      int2str(kp_tone) ' ...'])
k = [0:(K_SF-1)];
kr = [0:(K_SF/kp_tone):(K_SF-K_SF/(2*kp_tone))];
[junk, num_basis] = size(basis);
for b = 1:num_basis
   basisr(b,:) = spline(k,basis(:,b),kr);
   normfactor(b) = norm(basisr(b,:));
   basisr(b,:) = basisr(b,:)/norm(basisr(b,:)); % ensure 2-norm = 1
end
basisr = [basisr basisr];

%% Build basis function 1 varying-phase matrix
basis1vpm = zeros(kp_tone);
for k = 1:kp_tone
   basis1vpm(k,1:kp_tone) = basisr(1,k:k+kp_tone-1);
end
basis1vpm = [basis1vpm;basis1vpm];

%% Phase-align basis1 with beginning of sustain region
tone_window = tone(sus_beg:sus_beg+kp_tone-1);
c = basis1vpm(1:kp_tone,1:kp_tone) * tone_window;
[maxval,kphase] = max(c);  % maxval not used
bp = sus_beg;
clear c
%% Determine amplitude coefficients for this first phase value
amp_envs = (basisr(:,kphase:kphase+kp_tone-1) * tone_window)';

%% Compute phases and amp_envs from sus_beg to end of tone
k_end = length(tone);
period_int = 2;      % how often to estimate phase - every 2 periods
           % algorithm requires this value to be positive integer
k_search_win = round(kp_tone/16);   % search window defines largest
         % phase variations which algorithm will track over period_int
k_tone = sus_beg + period_int*kp_tone;
index = 1;
num_its_est = round((k_end-k_tone)/period_int/kp_tone);
disp (['Estimated number of sustain breakpoints to compute: ' ..
       num2str(num_its_est) ' ...'])
kphase = [kphase zeros(1,num_its_est)];
kphase_uw = [kphase zeros(1,num_its_est)];
amp_envs = [amp_envs; zeros(num_its_est, num_basis)];
bp = [bp zeros(1,num_its_est)];
while k_tone <= (k_end - kp_tone + 1)
   index = index + 1;
   bp(index) = k_tone;
   tone_window = tone(k_tone:k_tone+kp_tone-1);
   k_last = kphase(index-1);
   if k_last-k_search_win < 1
      k_last = k_last + kp_tone;
   end
   c = basis1vpm(k_last-k_search_win:k_last+k_search_win,1:kp_tone) ..
      * tone_window;
```

```
    [maxval,loc] = max(c);
    kphase(index) = k_last-k_search_win + loc-1;
    if kphase(index) >= kp_tone + 1
       kphase(index) = rem(kphase(index),kp_tone);
    end
    % Compute unwrapped phase value
    if abs(kphase(index) - kphase(index-1)) > k_search_win,
       % phase has wrapped
       if kphase(index) < kphase(index-1)
          % positive going phase wrap
          kphase_uw(index) = kphase_uw(index-1) + period_int*kp_tone ..
                             + kphase(index) + kp_tone - kphase(index-1);
       else
          % negative going phase wrap
          kphase_uw(index) = kphase_uw(index-1) + period_int*kp_tone ..
                             + kphase(index) - kp_tone - kphase(index-1);
       end
    else
       % phase hasn't wrapped
       kphase_uw(index) = kphase_uw(index-1) + period_int*kp_tone ..
                          + kphase(index) - kphase(index-1);
    end
    % Compute amplitude values
    amp_envs(index,:) = ..
      (basisr(:,kphase(index):kphase(index)+kp_tone-1) * tone_window)';
    k_tone = k_tone + period_int*kp_tone;
end
%D disp([max(kphase) min(kphase)])

%% Compute breakpoint/envelope values for end of tone
bp(index+1) = length(tone);
% Estimate final phase value based on previous 2 phase values
kphase_end = round(kphase(index) + (bp(index+1)-bp(index)) * ..
    (kphase_uw(index) - kphase_uw(index-1)) / (bp(index) - bp(index-1)));
while kphase_end > kp_tone
    kphase_end = kphase_end - kp_tone;
end
kphase(index+1) = kphase_end;
kphase_uw_end = round(kphase_uw(index) + (bp(index+1)-bp(index)) * ..
    (kphase_uw(index) - kphase_uw(index-1)) / (bp(index) - bp(index-1)));
kphase_uw(index+1) = kphase_uw_end;
amp_envs(index+1,:) = zeros(1,num_basis);

%% Store envelopes for sustain region
kphase_sus = kphase(1:index+1);
kphase_uw_sus = kphase_uw(1:index+1);
amp_envs_sus = amp_envs(1:index+1,:);
bp_sus = bp(1:index+1);

%% Compute phases and amp_envs from sus_beg to beginning of tone
%% Attack region
period_int = -0.5;    % how often to estimate phase - every 0.5 period
k_search_win = round(kp_tone/16);    % search window defines largest
    % phase variations which algorithm will track over period_int
k_tone = round(sus_beg + period_int*kp_tone);
index = 1;
num_its_est = round(k_tone/abs(period_int)/kp_tone);
```

```
disp (['Estimated number of attack breakpoints to compute: ' ..
       num2str(num_its_est) ' ...'])
kphase = [kphase_sus(1) zeros(1,num_its_est-1)];
kphase_uw = [kphase_uw_sus(1) zeros(1,num_its_est-1)];
amp_envs = [zeros(num_its_est, num_basis)];
bp = [bp zeros(1,num_its_est)];
while k_tone >= 1
   index = index + 1;
   bp(index) = k_tone;
   tone_window = tone(k_tone:k_tone+kp_tone-1);
   k_last = kphase(index-1);
   k_now = round(k_last + period_int*kp_tone);
   if k_now < 1
      k_now = k_now + kp_tone;
   end
   if k_now-k_search_win < 1
      k_now = k_now + kp_tone;
   end
   c = basis1vpm(k_now-k_search_win:k_now+k_search_win,1:kp_tone) ..
      * tone_window;
   [maxval,loc] = max(c);
   kphase(index) = k_now-k_search_win + loc-1;
   if kphase(index) >= kp_tone + 1
      kphase(index) = rem(kphase(index),kp_tone);
   end
   % Compute unwrapped phase value
   % Following is sufficient test for wrapping in case of sampling
   % every half-period
   if kphase(index) > kphase(index-1)
      % phase has wrapped
      kphase_uw(index) = kphase_uw(index-1) ..
                         - (kphase(index-1) + kp_tone - kphase(index));
   else
      % phase hasn't wrapped
      kphase_uw(index) = kphase_uw(index-1) - (kphase(index-1)- ..
         kphase(index));
   end
   amp_envs(index,:) = ..
     (basisr(:,kphase(index):kphase(index)+kp_tone-1) * tone_window)';
   k_tone = round(k_tone + period_int*kp_tone);
end
%D disp([max(kphase) min(kphase)])

%% Construct final output values
if bp(index) ~= 1
   bp = [1 fliplr(bp(2:index)) bp_sus];
   % Estimate first phase value based on next 2 phase values
   kphase_beg = round(kphase(index) - ..
    (bp(2)-bp(1)) * (kphase_uw(index-1) - kphase_uw(index)) / (bp(3) -
bp(2)));
   if kphase_beg < 1
      kphase_beg = kphase_beg + kp_tone;
   end
   kphase = [kphase_beg fliplr(kphase(2:index)) kphase_sus];
   kphase_uw_beg = round(kphase_uw(index) - ..
    (bp(2)-bp(1)) * (kphase_uw(index-1) - kphase_uw(index)) / ..
    (bp(3) - bp(2)));
```

```
   kphase_uw = [kphase_uw_beg fliplr(kphase_uw(2:index)) kphase_uw_sus];
   amp_envs = [zeros(1,num_basis); flipud(amp_envs(2:index,:));
amp_envs_sus];
else
   bp = [fliplr(bp(2:index)) bp_sus];
   kphase = [fliplr(kphase(2:index)) kphase_sus];
   kphase_uw = [fliplr(kphase_uw(2:index)) kphase_uw_sus];
   amp_envs= [zeros(1,num_basis); flipud(amp_envs(2:index-1,:)); ..
              amp_envs_sus];
end

% Scale amplitude envelopes for normalized basis functions
for b = 1:num_basis
   amp_envs(:,b) = amp_envs(:,b)/normfactor(b);
end

%% Construct frequency envelope
lbp = length(bp);
dphase = kphase_uw(2:lbp) - kphase_uw(1:lbp-1);
f_env = dphase ./ (bp(2:lbp) - bp(1:lbp-1)) / kp_tone * FS_PB;
f_env(lbp) = f_env(lbp-1);
f_env = f_env';
% Override first estimate
f_env(1) = median(f_env);

%% Smooth phase quantization effects in frequency envelope
smooth_win = hamming(SMOOTH_WIN_SIZE);
smooth_win = smooth_win'/sum(smooth_win);
halfwin = floor(SMOOTH_WIN_SIZE/2);

temp_env = [flipud(f_env(1:halfwin));
            f_env;
            flipud(f_env(lbp-halfwin+1:lbp))];
f_env_smooth = zeros(lbp,1);
for b = 1:lbp
   f_env_smooth(b) = smooth_win * temp_env(b:b+SMOOTH_WIN_SIZE-1);
end

%D plot(bp,f_env,bp,f_env_smooth)
%D keyboard

f_env = f_env_smooth;
```

4.3. Generate Amplitude Synthesis Parameters Subroutine

```
function [dys, dysbp, y] = amp_synthpars ..
   (amp_envs, bp, sus_beg, sus_end, fs_tone)
% m-func for extracting the scaled amplitude rate-of-change parameters
% at reduced number of breakpoints from an amplitude envelope.
%
% Call:  [dys, dysbp, y] = amp_synthpars ..
%        (amp_envs, bp, sus_beg, sus_end, fs_tone)
%
% Where: dys is a matrix containing the scaled delta amplitude
%           parameters, the ith column corresponding to the ith
```

```
%           basis function.
%        dysbp is a matrix containing the breakpoints corresponding
%           to the scaled delta amplitude parameters in dys.  Note
%           that the column length is 1 greater than for dys, the
%           last breakpoint marking the end of the last amplitude
%           segment.
%        y is a matrix containing the amplitude envelope values
%           corresponding to the breakpoints in dysbp, passed
%           back for review purposes.
%        amp_envs is a MxN matrix, the nth column containing the
%           amplitude envelope of the nth basis function.
%        bp is a length M vector containing the breakpoint sample
%           values for amp_envs.
%        sus_beg is the estimated beginning of sustain region.
%        sus_end is the estimated end of sustain region.
%        fs_tone is the sampling frequency of the original tone.
%
% Method:
%     1) Call ADJUST algorithm for reducing breakpoints, then
%        compute scaled delta amplitude parameters.
%     2) Call graphical user input algorithm as alternative.
%
% References:  Stapleton Thesis.
%              My notes.
%
% Rev. History:   Original Code: June 6, 1991
%     Modified: August 8, 1991
%     - pass back y to review analysis
%     Modified: August 13, 1991
%     - call amp_seg_userin to specify amplitude envelope
%       linear segments
%     Modified: August 21, 1991
%     - Use adjust, or env_seg_userin to specify amplitude
%       envelopes based on value of global USE_ADJUST
%     Modified: September 25, 1991
%     - do not plot if global BATCH_MODE ~= 0
%     - receives sus_end as a parameter
%     - Calls env_seg_userin2 for graphical input

% Calls:
% Matlab:   size, plot
%
% KLAnal:   adjust, env_seg_userin or env_seg_userin2

%% Compute synthpars for each basis function

%% First, segment amplitude envelopes into fewer breakpoints, using
%% either adjust algorithm, or graphical user input.
[junk,num_basis] = size(amp_envs);
if USE_ADJUST
   for b = 1:num_basis
      disp(['Segmenting amplitude envelope ' int2str(b) ' ...'])
      [y(:,b), dysbp(:,b), num_its] = adjust(amp_envs(:,b),bp, sus_beg);
      if ~BATCH_MODE
         clg
         plot(bp,amp_envs(:,b),dysbp(:,b),y(:,b),'o', ..
              dysbp(:,b),y(:,b),'--')
```

```
         title('Piecewise Linear Interpolation of Amplitude Envelope')
         xlabel(['Number of Iterations = ' int2str(num_its)])
         if PAUSE_ON_GRAPHS
            mypause
         end
      end
   end
else
   [y, dysbp] = env_seg_userin2 (amp_envs, bp, sus_beg, sus_end, ..
         'Amplitude');
end

%% Compute scaled delta amplitude parameters at new breakpoints
disp('Computing scaled amplitude synthesis parameters ...')
for b = 1:num_basis
   dy = y(2:NUM_SEGS+1,b) - y(1:NUM_SEGS,b);
   dbp = dysbp(2:NUM_SEGS+1,b) - dysbp(1:NUM_SEGS,b);
   dys(:,b) = (dy ./ dbp) * fs_tone/FS_PB;
end
```

4.3.1. ADJUST Subroutine

```
function [y, bp, num_its] = adjust (env, env_bp, sus_beg)
% m-func for performing the ADJUST algorithm for constructing best-fit
% piecewise linear interpolation of envelope
%
% Call:  [y, bp, num_its] = adjust (env, env_bp, sus_beg)
%
% Where: y is the column vector of the values of env pointed
%           to by the elements of bp
%        bp is the column vector of breakpoints
%        num_its is the number of iterations performed in improving
%           the piecewise fit
%        env is the envelope to piecewise interpolate
%        env_bp is the breakpoints for env
%        sus_beg is the estimated beginning of sustain region
%
% Method:   ADJUST algorithm
%
% References:  "Approximation and Syntactic Analysis of Amplitude and
%              Frequency Functions for Digital Sound Synthesis" by
%              John Strawn, in "The Music Machine"
%
% Rev. History:   Original Code: June 7, 1991
%     Modified: July 31, 1991
%       - maximum number of adjust iterations now specified by
%         global ADJUST_MAXITS

% Calls:
% Matlab:   length, size, rem, norm, min, max, table1

%% Initialize segments
M = 1;
bpi(1) = 1;
bpi(NUM_SEGS+1) = length(env_bp);
```

```
num_attack_bps = fix(2*NUM_SEGS/5); % 2/5 of bps initially in attack
[junk,bpi(num_attack_bps+1)] = min(abs(env_bp - sus_beg));
% Check for over-sampling short attack segments
if num_attack_bps > bpi(num_attack_bps+1)/4
   num_attack_bps = fix(bpi(num_attack_bps+1)/4);
   [junk,bpi(num_attack_bps+1)] = min(abs(env_bp - sus_beg));
end
for k = 2:num_attack_bps
   bpi(k) = fix((k-1)/num_attack_bps * bpi(num_attack_bps+1));
end
num_sus_env_bps = length(env_bp) - bpi(num_attack_bps+1);
for k = 1:NUM_SEGS-num_attack_bps-1
   bpi(k+num_attack_bps+1) = ..
    fix(k/(NUM_SEGS-num_attack_bps) * num_sus_env_bps) +
   bpi(num_attack_bps+1);
end

%% Perform ADJUST algorithm
bp_changed = 1;
bp_notchanged_once = 0;
num_its = 0;
notdone = 1;
while notdone
   num_its = num_its + 1;
   bp_changed = 0;
   if rem(num_its,2) == 1  % if odd iteration
      start = 1;
   else            % if even iteration
      start = 2;
   end
   for k = start:2:NUM_SEGS-1
      for seg = k:k+1
         yseg = env(bpi(seg):bpi(seg+1));
         tab = [env_bp(bpi(seg)) env_bp(bpi(seg+1)); ..
                env(bpi(seg)) env(bpi(seg+1))]';
         lseg = table1(tab,env_bp(bpi(seg):bpi(seg+1)));
         e(seg-k+1) = norm(lseg-yseg,2);
%O me        / (env_bp(bpi(seg+1)) - env_bp(bpi(seg)));
      end
      if e(1) ~= e(2)
         bppi = bpi;
         if e(1) > e(2)
            bppi(k+1) = bppi(k+1) - M;
         else
            bppi(k+1) = bppi(k+1) + M;
         end
         for seg = k:k+1
            yseg = env(bppi(seg):bppi(seg+1));
            tab = [env_bp(bppi(seg)) env_bp(bppi(seg+1)); ..
                   env(bppi(seg)) env(bppi(seg+1))]';
            lseg = table1(tab,env_bp(bppi(seg):bppi(seg+1)));
            ep(seg-k+1) = norm(lseg-yseg,2);
%O me        / (env_bp(bpi(seg+1)) - env_bp(bpi(seg)));
         end
         if max([e(1) e(2)]) > max([ep(1) ep(2)])
            bpi(k+1) = bppi(k+1);
            bp_changed = 1;
```

```
            bp_notchanged_once = 0;
         end
      end
   end
   if ~bp_changed
      if bp_notchanged_once
         notdone = 0;
      else
         bp_notchanged_once = 1;
      end
   end
   if num_its == ADJUST_MAXITS
      notdone = 0;
   end
end

%% Setup y and bp, Ensure return of column vectors
y = env(bpi);
[rows,cols] = size(y);
if rows == 1,
   y = y';
end
bp = env_bp(bpi);
[rows,cols] = size(bp);
if rows == 1,
   bp = bp';
end
```

4.3.2. User Selects Envelope Approximation Subroutine Type 1

```
function [y, dysbp] = env_seg_userin (envs, bp, env_type)
% m-func for allowing user to graphically select linear segments to
% approximate envelope
%
% Call:  [y, dysbp] = env_seg_userin (envs, bp, env_type)
%
% Where: dys is a matrix containing the amplitude
%           parameters, the ith column corresponding to the ith
%           basis function.
%        dysbp is a matrix containing the breakpoints corresponding
%           to the scaled delta amplitude parameters in dys.
%        envs is a MxN matrix, the nth column containing the
%           amplitude envelope of the nth basis function.
%        bp is a length M vector containing the breakpoint sample
%           values for envs and kphase.
%        env_type = 'Amplitude', amplitude envelope,
%                 = 'Frequency', frequency envelope
%
% Method:   User selects breakpoints
%
% Rev. History:   Original Code: Spring, 1991
%     Modified: August 21, 1991
%     - segment frequency or amplitude envelopes, add parameter
%       env_type to specify which, changed name to env_seg_userin
%       from amp_seg_userin.  Also, uses global NUM_SEGS to set
%       number of segments, therefore, no need for special first
```

```
%       pass.

% Calls:
% Matlab:   disp, plot, ginput, min

%% Initialize matrices dysbp and y
numdysbp = NUM_SEGS + 1;
[junk, numbasis] = size(envs);
dysbp = ones(numdysbp,numbasis);
y = zeros(numdysbp,numbasis);
y(1,:) = envs(1,:);
numbp = length(bp);
for b = 1:numbasis
   dysbp(numdysbp,b) = bp(numbp);
end
y(numdysbp,:) = envs(numbp,:);

%% Get user to enter breakpoints for each envelope
for b = 1:numbasis
   clg
   plot(bp,envs(:,b))
   title([env_type ' Envelope of Basis Function ' int2str(b)])
   disp(['Choose ' int2str(numdysbp-2) ..
         ' breakpoints with mouse - terminates automatically'])
   disp('(Start and End breakpoints will be generated automatically)')
   hold on
   for k = 2:numdysbp-1
      [x,junk] = ginput(1);
      [diff,locbp] = min(abs(bp-x));   % Determine exact breakpoint
      dysbp(k,b) = bp(locbp);
      y(k,b) = envs(locbp,b);
      plot(dysbp(k-1:k,b),y(k-1:k,b),'o')
      plot(dysbp(k-1:k,b),y(k-1:k,b),'--')
      disp(['Breakpoint ' int2str(k-1) ' of ' ..
         int2str(numdysbp-2) ' entered'])
   end
   k = k+1;
   plot(dysbp(k-1:k,b),y(k-1:k,b),'o')
   plot(dysbp(k-1:k,b),y(k-1:k,b),'--')
   hold off
   mypause
end
```

4.3.3. User Selects Envelope Approximation Subroutine Type 2

```
function [y, dysbp] = env_seg_userin2 (envs, bp, sus_beg, sus_end, ..
   env_type)
% m-func for allowing user to graphically select linear segments to
% approximate envelope
%
% Call:  [y, dysbp] = env_seg_userin2 ..
%        (envs, bp, sus_beg, sus_end, env_type)
%
% Where: dys is a matrix containing the amplitude
%           parameters, the ith column corresponding to the ith
%           basis function.
```

```
%        dysbp is a matrix containing the breakpoints corresponding
%           to the scaled delta amplitude parameters in dys.
%        envs is a MxN matrix, the nth column containing the
%           amplitude envelope of the nth basis function.
%        bp is a length M vector containing the breakpoint sample
%           values for envs and kphase.
%        sus_beg is the estimated beginning of sustain region.
%        sus_end is the estimated end of sustain region.
%        env_type = 'Amplitude', amplitude envelope,
%                 = 'Frequency', frequency envelope
%
% Method:   User selects breakpoints

% Rev. History:   Original Code: Spring, 1991
%     Modified: August 21, 1991
%     - segment frequency or amplitude envelopes, add parameter
%       env_type to specify which, changed name to env_seg_userin
%       from amp_seg_userin.  Also, uses global NUM_SEGS to set
%       number of segments, therefore, no need for special first
%       pass.
%     Modified: September, 1991
%     - oops, didn't leave a comment for this - something to do
%       with using NUM_ATTACK_SEGS and NUM_DECAY_SEGS instead of
%       NUM_SEGS to give more explicit attack and decay resolution -
%       only one sustain segment.

% Calls:
% Matlab:   disp, plot, ginput, min

%% Initialize matrices dysbp and y - including beginning and end
%% of envelopes
numdysbp = NUM_ATTACK_SEGS + NUM_DECAY_SEGS + 2;
[junk, numbasis] = size(envs);
dysbp = ones(numdysbp,numbasis);
y = zeros(numdysbp,numbasis);
y(1,:) = envs(1,:);
numbp = length(bp);
for b = 1:numbasis
   dysbp(numdysbp,b) = bp(numbp);
end
y(numdysbp,:) = envs(numbp,:);

%% Get user to enter breakpoints for each envelope
[junk,sus_beg_bpi] = min(abs(bp-sus_beg));
[junk,sus_end_bpi] = min(abs(bp-sus_end));
for b = 1:numbasis
   clg
   plot(bp,envs(:,b))
   title([env_type ' Envelope of Basis Function ' int2str(b)])
   hold on
   % Compute and plot sustain envelope
   sus_level = mean(envs(sus_beg_bpi:sus_end_bpi,b));
   y(NUM_ATTACK_SEGS+1,b) = sus_level;
   dysbp(NUM_ATTACK_SEGS+1,b) = bp(sus_beg_bpi);
   y(NUM_ATTACK_SEGS+2,b) = sus_level;
   dysbp(NUM_ATTACK_SEGS+2,b) = bp(sus_end_bpi);
   k = NUM_ATTACK_SEGS + 2;
```

```
   plot(dysbp(k-1:k,b),y(k-1:k,b),'o')
   plot(dysbp(k-1:k,b),y(k-1:k,b),'--')
   % Query user for attack
   disp(['Choose ' int2str(NUM_ATTACK_SEGS-1) ..
         ' Attack breakpoints with mouse - terminates automatically'])
   disp('(Start breakpoint will be generated automatically)')
   for k = 2:NUM_ATTACK_SEGS
      [x,junk] = ginput(1);
      [diff,locbp] = min(abs(bp-x));   % Determine exact breakpoint
      dysbp(k,b) = bp(locbp);
      y(k,b) = envs(locbp,b);
      plot(dysbp(k-1:k,b),y(k-1:k,b),'o')
      plot(dysbp(k-1:k,b),y(k-1:k,b),'--')
      disp(['Breakpoint ' int2str(k-1) ' of ' ..
         int2str(NUM_ATTACK_SEGS-1) ' entered'])
   end
   k = k+1;
   plot(dysbp(k-1:k,b),y(k-1:k,b),'o')
   plot(dysbp(k-1:k,b),y(k-1:k,b),'--')
   % Query user for decay
   disp(['Choose ' int2str(NUM_DECAY_SEGS-1) ..
         ' Decay breakpoints with mouse - terminates automatically'])
   disp('(End breakpoint will be generated automatically)')
   for k = NUM_ATTACK_SEGS+3:numdysbp-1
      [x,junk] = ginput(1);
      [diff,locbp] = min(abs(bp-x));   % Determine exact breakpoint
      dysbp(k,b) = bp(locbp);
      y(k,b) = envs(locbp,b);
      plot(dysbp(k-1:k,b),y(k-1:k,b),'o')
      plot(dysbp(k-1:k,b),y(k-1:k,b),'--')
      disp(['Breakpoint ' int2str(k-NUM_ATTACK_SEGS-2) ' of ' ..
       int2str(NUM_DECAY_SEGS-1) ' entered'])
   end
   k = k+1;
   plot(dysbp(k-1:k,b),y(k-1:k,b),'o')
   plot(dysbp(k-1:k,b),y(k-1:k,b),'--')
   hold off
   mypause
end
```

4.4. Generate Frequency Synthesis Parameters Subroutine

```
function [dfs, dfsbp, f] = freq_synthpars(f_env, bp, sus_beg, fs_tone)
% m-func for extracting the scaled frequency rate-of-change parameters
% at reduced number of breakpoints from a phase envelope.
%
% Call:  [dfs, dfsbp, f] = freq_synthpars(kphase, bp, sus_beg, fs_tone)
%
% Where: dfs is a vector containing the scaled delta frequency
%           parameters.
%        dfsbp is a vector containing the breakpoints corresponding
%           to the scaled delta frequency parameters in dfs.  Note
%           that the column length is 1 greater than for dfs, the
%           last breakpoint marking the end of the last frequency
%           segment.
%        f is a vector containing the frequency envelope values
```

```
%           corresponding to dfsbp, passed back for review.
%        f_env is a length M vector containing the frequency
%           envelope.
%        bp is a length M vector containing the breakpoint sample
%           values for f_env.
%           sus_beg is the estimated beginning of sustain region
%           fs_tone is the sampling frequency of the original tone.
%
% Method:   First generate frequency envelope from kphase, then
%           call ADJUST algorithm for reducing breakpoints, then
%           compute scaled delta frequency parameters.
%
% References:  Stapleton Thesis.
%              My notes.
%
% Rev. History:   Original Code: June 10, 1991
%     Modified: August 8, 1991
%     - pass back f for graphical review of analysis
%     Modified: August 13, 1991
%     - 1 segment, DC frequency "envelope"
%     Modified: August 21, 1991
%     - responds to globals FLAT_FREQUENCY and USE_ADJUST
%       to determine how frequency envelope is generated:
%       FLAT_FREQUENCY   USE_ADJUST response
%             0              0        call env_seg_userin
%             0              1        call adjust
%             1          don't care   1 segment flat frequency
%     Modified: September 25, 1991
%     - do not plot if global BATCH_MODE ~= 0

% Calls:
% Matlab:   disp, plot
%
% KLAnal:   adjust, env_seg_userin

%% Compute synthpars for each basis function
%% First, segment frequency envelope into fewer breakpoints
disp(['Segmenting frequency envelope ...'])
if FLAT_FREQUENCY
   f(1) = median(f_env);
   f(2) = f(1);
   dfsbp = [bp(1) bp(length(bp))];
   if ~BATCH_MODE
      clg
      plot(bp,f_env,dfsbp,f,'o',dfsbp,f,'--')
      title('Flat Frequency Envelope')
      if PAUSE_ON_GRAPHS
         mypause
      end
   end
else
   if USE_ADJUST
      [f, dfsbp, num_its] = adjust(f_env, bp, sus_beg);
      if ~BATCH_MODE
         clg
         plot(bp,f_env,dfsbp,f,'o',dfsbp,f,'--')
         title('Piecewise Linear Interpolation of Frequency Envelope')
```

```
         xlabel(['Number of Iterations = ' int2str(num_its)])
         if PAUSE_ON_GRAPHS
            mypause
         end
      end
   else
      [f, dfsbp] = env_seg_userin (f_env, bp, 'Frequency');
   end
end

%% Compute scaled delta frequency parameters at new breakpoints
disp('Computing scaled frequency synthesis parameters ...')
if FLAT_FREQUENCY
   df = f(2) - f(1);
   dbp = dfsbp(2) - dfsbp(1);
else
   df = f(2:NUM_SEGS+1) - f(1:NUM_SEGS);
   dbp = dfsbp(2:NUM_SEGS+1) - dfsbp(1:NUM_SEGS);
end
dfs = (df ./ dbp / fs_tone * K_SF) * (fs_tone/FS_PB)^2;
```

5. Review Synthesis Parameter Extraction - Main Routine

```
function review_synthpars
% m-func for reviewing the synthesis parameter analysis
%
% Call:  review_synthpars
%
% Where: No input or output parameters.
%
% Method:   Plot graphs which were presented in extract_synthpars
%
% Rev. History:   Original Code: August 8, 1991
%     Modified: August 24, 1991
%     - loads all parameters for review from one file

% Calls:
% Matlab:   disp, eval, dir, int2str, input, isempty, length
%     plot etc.
%
% KLAnal:   mypause

%% Choose Ensemble, then Instrument Tone to review, and load
%% Instrument Tone synthesis parameters for review
notdone = 1;
while notdone
   disp('Analyzed Ensembles:')
   eval(['dir ' DIR_SYNTHPARS int2str(K_SF)])
   ens_name=input ..
    ('Choose Ensemble for Analysis Review (<CR> to Quit): ','s');
   if isempty(ens_name)
      return;
   end
   disp('Instrument Tones in Ensemble:')
   eval(['dir ' DIR_SYNTHPARS int2str(K_SF) '/' ens_name]);
```

```
   tone_name = input('Choose Instrument Tone for Review: ','s');
   % Strip off .mat if included
   if length(tone_name) > 4
      if tone_name(length(tone_name)-3:length(tone_name)) == '.mat'
         tone_name = tone_name(1:length(tone_name)-4);
      end
   end
   % Load synthesis parameters file
   filename = [DIR_SYNTHPARS int2str(K_SF) '/' ens_name '/' ..
               tone_name '.mat'];
   if exist(filename) == 2
      disp(['Loading ' tone_name '_rev.mat ...'])
      eval(['load ' filename])
      disp('Synthesis parameters loaded')
      notdone = 0;
   else
      disp(['Can''t find ' filename])
   end
end
%% Synthesis parameters loaded:
%% y_init dys dysbp kphase_init fs_init dfs dfsbp basis fs_pb
%% bp amp_envs f_env kphase kphase_uw fs kp_tone y f

%% Plot amplitude, phase and frequency envelopes
clg
subplot(221),plot(bp,kphase),title('Time-varying Phase (Wrapped)')
subplot(222),plot(bp,amp_envs), ..
  title('Basis Function Amplitude Envelopes')
subplot(223),plot(bp,kphase_uw),title('Unwrapped Phase')
disp(['Estimated frequency of tone = ' num2str(fs/kp_tone) ' Hz'])
subplot(224),plot(bp,f_env),title('Frequency Envelope')
mypause
%D keyboard

%% Plot segmented amplitude envelopes
[junk,num_basis] = size(amp_envs);
for b = 1:num_basis
   clg
   plot(bp,amp_envs(:,b),dysbp(:,b),y(:,b),'o',dysbp(:,b),y(:,b),'--')
   title('Piecewise Linear Interpolation of Amplitude Envelope')
   mypause
end

%% Plot segmented frequency envelope
plot(bp,f_env,dfsbp,f,'o',dfsbp,f,'--')
title('Piecewise Linear Interpolation of Frequency Envelope')
```

6. Re-synthesize a Tone in Non-Realtime - Main Routine

```
function re_synth
% m-func for re-synthesizing tone from synthesis parameters,
% in non-realtime.
%
% Call:     re_synth
%
```

```
% Where: No Input or Output parameters
%
% Method:   See code.
%
% References:  Stapleton Thesis.
%              My Notes.
%
% Rev. History:   Original Code: June 25, 1991
%     Modified: August 13, 1991
%     - assumes single segment DC or linear frequency envelope
%     Modified: August 21, 1991
%     - checks for either single or full segment frequency
%       envelope (full = same # of segments as for amp_envs)
%     Modified: August 22, 1991
%     - check for tones with sample values greater than 16-bit
%       integer range, and scale tone, if so.
%     - allow user to specify less than full number of basis
%       functions for synthesis.

% Calls:
% Matlab:   disp, eval, dir, input, isempty, length, exist, size
%           zeros, find, max, min, abs, round, int2str, load, save
%
% KLAnal:   additive

%% Choose Ensemble, then Instrument Tone to re-synthesize, and load
%% Instrument Tone synthesis parameters
notdone = 1;
while notdone
   disp('Analyzed Ensembles:')
   eval(['dir ' DIR_SYNTHPARS int2str(K_SF)])
   ens_name = input ..
      ('Choose Ensemble for Re-Synthesis (<CR> to Quit): ','s');
   if isempty(ens_name)
      return;
   end
   disp('Instrument Tones in Ensemble:')
   eval(['dir ' DIR_SYNTHPARS int2str(K_SF) '/' ens_name]);
   tone_name = input('Choose Instrument Tone for Re-Synthesis: ','s');
   % Strip off .mat if included
   if length(tone_name) > 4
      if tone_name(length(tone_name)-3:length(tone_name)) == '.mat'
    tone_name = tone_name(1:length(tone_name)-4);
      end
   end
   filename = [DIR_SYNTHPARS int2str(K_SF) '/' ens_name ..
               '/' tone_name '.mat'];
   if exist(filename) == 2
      disp(['Loading ' tone_name '.mat ...'])
      eval(['load ' filename])
      disp('Tone synthesis parameters loaded')
      notdone = 0;
   else
      disp(['Can''t find ' filename])
   end
end
%% Synthesis parameter variables loaded:
```

```
%% dys, dysbp, y_init, dfs, dfsbp, fs_init, kphase_init, basis, fs_pb
%% bp, amp_envs, f_env, kphase, kphase_uw, fs, kp_tone, y, f
clear bp amp_envs f_env kphase kphase_uw fs kp_tone y f

%% Synthesize tone
% Ask user how many of the basis functions to use?
[junk, num_basis] = size(dysbp);
num_amp_envs = 0;
while (num_amp_envs < 1) | (num_amp_envs > num_basis)
   disp([int2str(num_basis) ' basis function envelopes available.'])
   num_amp_envs = input ..
   ('Enter number of basis functions to use for synthesis: ');
end
dys = dys(:,1:num_amp_envs);
dysbp = dysbp(:,1:num_amp_envs);
y_init = y_init(:,1:num_amp_envs);

% Initial settings
if length(dfsbp) > 2
   bpmatrix = [dysbp dfsbp];  % matrix of breakpoints
   [bprows, bpcols] = size(bpmatrix);
else
   bpmatrix = [dysbp];     % matrix of amplitude breakpoints
   [bprows, bpcols] = size(bpmatrix);
end
bpi = zeros(1,bpcols) + 2; % point to all 2nd breakpoints
tone = zeros(1,max(max(bpmatrix))); % initialize tone vector
tone(1) = 0;
y_curr = y_init;
dys_curr = dys(1,:);
fs_curr = fs_init;
dfs_curr = dfs(1);
kphase_curr = kphase_init;
bp_last = 1;
bp_curr = min(bpmatrix(2,:));
bp_queue = bpmatrix(2,:);
basis = [basis(:,1:num_amp_envs) ; basis(1,1:num_amp_envs)];
   % wrap-around table
% Do the synthesis
notdone = 1;
   disp([bp_queue bp_curr; bpi 0])
while notdone
   [tone(bp_last+1:bp_curr),y_next,fs_next,kphase_next] = additive ..
     (basis,y_curr,dys_curr,fs_curr,dfs_curr,kphase_curr, ..
      bp_curr-bp_last);
   % Update appropriate envelope(s) and determine next breakpoint
   [envs] = find(bp_queue == bp_curr);
   for env = envs % for each envelope changing at breakpoint
      if env <= num_amp_envs  % all amplitude envelopes
         if bpi(env) < bprows % if not at end of breakpoints
            dys_curr(env) = dys(bpi(env),env);
         end
      else     % frequency envelope
         if bpi(env) < bprows % if not at end of breakpoints
            dfs_curr = dfs(bpi(env));
         end
```

```
      end
      bpi(env) = bpi(env) + 1;
      if bpi(env) <= bprows
         bp_queue(env) = bpmatrix(bpi(env),env);
      end
   end
   [envs_complete] = find(bpi > bprows);
   if length(envs_complete) == bpcols
      notdone = 0;
   else
      % update bp_curr and other parameters
      bp_last = bp_curr;
      bp_curr = min(bp_queue);
      y_curr = y_next;
      fs_curr = fs_next;
      kphase_curr = kphase_next;
   end
   disp([bp_queue bp_curr; bpi 0])
end

%% Check if 16 bit integer magnitude exceeded - report and scale if so
while (max(round(tone)) > 32767) | (min(round(tone)) < -32768)
   disp('16 bit integer magnitude exeeded - scaling tone ...')
   tone = 32767/max(abs(round(tone))) * tone;
end

%% Save synthesized tone
% As Matlab file data file
syntone = round(tone);
[junk,numbasis] = size(dys);
% tag on number of basis functions to tone_name
if length(tone_name) == length(ens_name)
   if tone_name ~= ens_name
      % tag on ens_name if different from tone_name
      tone_name = [tone_name '_' int2str(numbasis) 'b_' ens_name];
   else
      tone_name = [tone_name '_' int2str(numbasis) 'b'];
   end
else
   tone_name = [tone_name '_' int2str(numbasis) 'b_' ens_name];
end
disp('Saving Synthesized Tone as Matlab file:')
disp(['  ' DIR_SYNTHTONES int2str(K_SF) '/' tone_name])
eval(['save ' DIR_SYNTHTONES int2str(K_SF) '/' tone_name ' ..
      syntone fs_pb'])
% As ASCII data file
syntoneint = round(tone');
fs_pbint = round(fs_pb);
num_samps = length(syntoneint);
disp('Saving Synth Tone as ASCII Data file for conversion to AIFF file')
disp(['  ' DIR_SYNTHTONES int2str(K_SF) '/' tone_name '.dat'])
eval(['save ' DIR_SYNTHTONES int2str(K_SF) '/' tone_name ..
      '.dat fs_pbint num_samps syntoneint /ascii /double'])
```

6.1. Additive Synthesis Subroutine

```
function [tone_seg, yn, fsn, kpn] = ..
  additive (basis, yc, dysc, fsc, dfsc, kpc, n_samp)
% m-func for synthesizing tone segment between envelope breakpoints
%
% Call:  [tone_seg, yn, fsn, kpn] = ..
%        additive (basis, yc, dysc, fsc, dfsc, kpc, n_samp)

% Initialize
tone_seg = zeros(1:n_samp);
num_osc = length(yc);
dfscby2 = dfsc/2;
% Synthesize
for n = 1:n_samp
   yc = yc + dysc;
   fsc = fsc + dfsc;
   kpc = kpc + fsc - dfscby2;
   if kpc >= K_SF+1
      kpc = kpc - K_SF;
   end
   kpcfix = fix(kpc);
   range1 = basis(kpcfix,:);
   range2 = basis(kpcfix+1,:);
   tone_seg(n) = sum(yc .* (range1 + (range2-range1)*(kpc-kpcfix)));
end
% Pass back current synthesis state for next run.
yn = yc;
fsn = fsc;
kpn = kpc;
```

7. Graphically Compare Original and Synthesized Tones - Main Routine

```
function compare
% m-func for compare synthesized tone with orginal tone graphically
%
% Call:     compare
%
% Where: No Input or Output parameters
%
% Method:   See code.
%
% Rev. History:   Original Code: July 3, 1991

%% Choose Synthesized Tone - load syntone and fs_pb variables
disp('Synthesized Tones:')
eval(['dir ' DIR_SYNTHTONES int2str(K_SF) '/*.mat'])
notdone = 1;
while notdone
   syn_name = input('Choose Synthesized Tone (<CR> to Quit): ','s');
   if isempty(syn_name)
      return;
   end
   % Strip off .mat if included
```

```
   if length(syn_name) > 4
      if syn_name(length(syn_name)-3:length(syn_name)) == '.mat'
         syn_name = syn_name(1:length(syn_name)-4);
      end
   end
   filename = [DIR_SYNTHTONES int2str(K_SF) '/' syn_name '.mat'];
   if exist(filename) == 2
      disp(['Loading ' syn_name '.mat ...'])
      eval(['load ' filename])
      disp('Synthesized tone loaded')
      notdone = 0;
   else
      disp(['Can''t find ' filename])
   end
end
%% Choose Instrument Tone - load tone and fs variables
disp('Original Instrument Tones:')
eval(['dir ' DIR_TONES '*.mat'])
notdone = 1;
while notdone
   tone_name = input('Choose Instrument Tone (<CR> to Quit): ','s');
   if isempty(tone_name)
      return;
   end
   % Strip off .mat if included
   if length(tone_name) > 4
      if tone_name(length(tone_name)-3:length(tone_name)) == '.mat'
         tone_name = tone_name(1:length(tone_name)-4);
      end
   end
   filename = [DIR_TONES tone_name '.mat'];
   if exist(filename) == 2
      disp(['Loading ' tone_name '.mat ...'])
      eval(['load ' filename])
      disp('Instrument tone loaded')
      notdone = 0;
   else
      disp(['Can''t find ' filename])
   end
end
%% Load Instrument Tone Sample Function File for needed variables
%%   variables to be loaded: samp_func, kp_tone, sus_beg, sus_end
%%   variables needed: kp_tone, sus_beg, sus_end
if exist([DIR_SFS int2str(K_SF) '/' tone_name '_sf.mat']) == 2
   disp(['Loading ' tone_name '_sf.mat ...'])
   eval(['load ' DIR_SFS int2str(K_SF) '/' tone_name '_sf'])
else
   disp(['Can''t find ' DIR_SFS int2str(K_SF) '/' tone_name '_sf.mat'])
   disp(['Exitting ' compareLN])
   return
end
% clear unused variables which were loaded
clear samp_func

disp(['Synthesized Tone Sample Rate = ' num2str(fs_pb)])
disp(['Original Instrument Tone Sample Rate = ' num2str(fs)])
disp(['Length of Synthesized Tone = ' int2str(length(syntone))])
```

```
disp(['Length of Original Instrument Tone = ' int2str(length(tone))])

%% Plot regions
NP = 4;
clg
subplot(211)
plot(tone(1:NP*kp_tone)),title('Original Tone Attack')
subplot(212)
plot(syntone(1:NP*kp_tone)),title('Synthesized Tone Attack')
mypause
clg
start = sus_beg;
k = [start:start+NP*kp_tone];
subplot(211)
plot(k,tone(start:start+NP*kp_tone)),title('Original Tone Sustain
Begin')
subplot(212)
plot(k,syntone(start:start+NP*kp_tone)),title('Synthesized Tone Sustain
Begin')
mypause
clg
start = fix(mean([sus_beg sus_end]));
k = [start:start+NP*kp_tone];
subplot(211)
plot(k,tone(start:start+NP*kp_tone)),title('Original Tone Sustain')
subplot(212)
plot(k,syntone(start:start+NP*kp_tone)),title('Synthesized Tone
Sustain')
mypause
clg
start = sus_end - NP*kp_tone;
k = [start:start+NP*kp_tone];
subplot(211)
plot(k,tone(start:start+NP*kp_tone)),title('Original Tone Sustain End')
subplot(212)
plot(k,syntone(start:start+NP*kp_tone)),title('Synthesized Tone Sustain
End')
mypause
clg
start = min([length(syntone) length(tone)]) - NP*kp_tone;
k = [start:start+NP*kp_tone];
subplot(211)
plot(k,tone(start:start+NP*kp_tone)),title('Original Tone Decay')
subplot(212)
plot(k,syntone(start:start+NP*kp_tone)),title('Synthesized Tone Decay')
mypause
```

VITA

Surname:	*Weeks*	Given Names:	*William Brent*
Place of Birth:	*Victoria, B.C.*	Date of Birth:	*February 3, 1965*

Educational Institutions Attended:

University of Victoria	1983 to 1992
Victoria Conservatory of Music	1983 to 1984

Degrees and Diplomas Awarded:

B.Eng.	University of Victoria	1989
A.R.C.T.	Royal Conservatory of Music	1984

Honours and Awards:

NSERC Post-Graduate Scholarship	1989-1991
President's Research Scholarship	1989-1991
CSECE Medal	1989

Publications:

W.B. Weeks, W.A. Schloss and R.L. Kirlin. "Implementation of the KL Synthesis Algorithm under Real-Time Control," *Proceedings of the 1991 International Computer Music Conference*, pp. 360-363, October 1991.

W.B. Weeks, W.D. Little, V.K. Bhargava and T.A. Gulliver. "Comparison of the Motorola DSP56000 and the Texas Instruments TMS320C25 Digital Signal Processors for Implementing a Four Error Correcting (127,99) BCH Error Control Code Decoder," *Proceedings of the 1989 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 345-349, June 1989.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request fom the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Refinements to the Analysis and Synthesis of Musical Tones using the Karhunen-Loève Transform

Author: [redacted]

William Brent Weeks

September 21, 1992